

ESTRUTURA DE DADOS E TÉCNICAS DE PROGRAMAÇÃO

VISIT...

LANZAROTE
Caliente.COM



ESTRUTURA DE DADOS E TÉCNICAS DE PROGRAMAÇÃO

I. ed.

Dilermando Piva Junior
Gilberto Shigueo Nakamiti
Ricardo Luís de Freitas
Leandro Alonso Xastre
Francisco Bianchi



© 2014, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Copidesque: Andréa Vidal de Miranda

Revisão: Casa Editorial BBM

Editoração Eletrônica: Thomson Digital

Elsevier Editora Ltda. Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 – 16º andar

20050-006 – Centro – Rio de Janeiro – RJ – Brasil

Rua Quintana, 753 – 8º andar

04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente

0800-026-5340 atendimento1@elsevier.com

ISBN: 978-85-352-7437-0

ISBN (versão digital): 978-85-352-7438-7

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO NA PUBLICAÇÃO
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

E85

Estrutura de dados e técnicas de programação / Dilermando Piva Junior ... [et al.]. - 1. ed. - Rio de Janeiro : Elsevier, 2014.

24 cm.

ISBN 978-85-352-7437-0

1. Estruturas de dados (Computação). 2. Programação (Computadores). I. Piva Junior, Dilermando. II. Título.

14-10204

CDD: 005.1

CDU: 004.42

*Essa obra é dedicada aos nossos mestres, que semearam em nossas mentes
o desejo incansável pela busca contínua do conhecimento. Aos nossos alunos,
com a esperança de que herdem com alegria e perseverança
essa responsabilidade e a propaguem.*

Agradecimentos

Agradecemos às nossas famílias, que sempre nos apoiaram, incondicionalmente.

Prefácio

Entre as áreas do conhecimento e as atividades humanas, talvez nenhuma supere a Computação quanto à convergência entre simplicidade e complexidade. Desde seu início, a computação tem-se revelado como a busca de caminhos simples e eficientes para resolver desafios amplos, diversos e complexos. Essa lógica se ajusta com perfeição ao tratamento das estruturas de dados enfeitado neste livro.

A complexidade abriga-se no tema. Mesmo considerando o conhecimento de uma linguagem de programação suficientemente para a construção de algoritmos que respondam a determinados problemas, é preciso não esquecer que a solução de problemas mais complexos exige o conhecimento e a aplicação da teoria de estruturas de dados, ainda que se tenha um conhecimento avançado de todos os recursos que uma linguagem oferece.

Não é por outra razão que o assunto constitui elemento fundamental nos currículos dos Cursos de Computação. Assim retratado pela primeira vez em 1968, lembrando que Donald E. Knuth desponta como pioneiro na apresentação e tratamento coerente das estruturas de dados lineares (listas, filas e pilhas) e as não lineares (árvores e grafos), tanto na representação sequencial quanto naquela classificada como ligada.

Neste livro, o assunto é oportunamente retomado para atender demandas e expectativas da comunidade acadêmica e de todos quantos se interessam pelo conhecimento teórico da Computação. Donos de admiração pela competência atestada curricularmente, têm os autores meu carinho, enquanto colegas docentes na PUC-Campinas, bem como minha admiração, pelo modo simples e, por isso mesmo, tão claro e compreensível que desenvolveram para tratar a imensa complexidade que se aloja no estudo e no entendimento das estruturas de dados.

Optando por caminho diverso, os autores transformam escrita em conversa e, já nas primeiras páginas, conseguem engajar o leitor em uma dinâmica troca de ideias, consolidando intuitivamente a compreensão do tema, à medida que a posição ativa de interlocutor substitui a condição passiva de receptor.

Além de qualificar a forma, a simplicidade também alinhava o conteúdo, em especial quanto a situações, analogias e exemplos apresentados ao leitor. Estimulando a

reflexão sobre funções e características das estruturas de dados sob a ótica de coisas e fatos do cotidiano, os autores transformam assuntos triviais em ferramentas indutivas e muito eficientes para acionar as engrenagens cognitivas do leitor e, por assim dizer, construir, camada após camada, mutuamente apoiadas, o entendimento do tema.

Seja pelo tema de que trata, seja pelo modo como o faz, o livro certamente vai despertar interesse de alunos e professores de Computação. Para aqueles que ensinam, cabe assinalar o modo muito peculiar e apropriado para o ambiente de sala de aula que a obra estabelece entre teoria e prática. Não se trata, portanto, de uma relação consequencial de teoria, apresentada *a priori*, seguida de possíveis aplicabilidades práticas. Buscando horizontes pedagógicos mais eficientes, o livro força um diálogo interativo entre o campo teórico e o terreno da prática. Trata-se de compreender a teoria a partir da observação funcional (operativa) da realidade. E para aqueles que aprendem com este livro, registre-se a facilidade e o envolvimento no processo de ensino-aprendizagem, possibilitando-lhes a aquisição de profundo conhecimento sobre o tema.

Priorizando, ainda, a indicação desta obra para professores de Computação, cabe ressaltar que, aqui, a simplicidade caracterizadora da linguagem não significa que os temas sejam apenas superficialmente tangidos. Ao contrário, raciocinar por veredas mais objetivas significa caminhar mais à frente em direção à essência do assunto, permitindo destrinchar a complexidade das teorias que dão conta das estruturas de dados. Seja porque os autores são conhecedores aquilatados do tema, seja porque a obra tem viés assumidamente pedagógico, seja, enfim, porque a compreensão verdadeira de tudo quanto nos cerca passa, obrigatoriamente, pelos domínios da teoria, nestas páginas simplicidade não é sinônimo de epidérmico, nem antônimo de complexo, ao mesmo tempo em que prática e teoria não se digladiam, completam-se.

Animado por exercícios e desafios apresentados ao leitor de modo muito próximo à estrutura de fases e passos próprios dos jogos eletrônicos, o livro transpira o universo da Informática em todas as páginas, incluindo endereços e sugestões que remetem o leitor ao ambiente virtual, para que possa ampliar suas fontes de informação e conhecer mais sobre os temas abordados no livro.

Navegar pelas páginas certamente vai propiciar ao leitor engajamento em um diálogo estimulante resultando em mais conhecimento sobre um tema fundamental da área, que se torna mais interessante e intrigante tanto mais dele sabemos e nele atuamos.

PROFA. DRA. ANGELA DE MENDONÇA ENGELBRECHT

*Reitora da PUC-Campinas e professora titular de Computação
do Centro de Ciências Exatas, Ambientais e de Tecnologias (CEATEC).
Graduada e Mestre em Ciência da Computação pela Universidade de São Paulo (USP)
e doutora em Engenharia Elétrica pela Universidade Estadual de Campinas (UNICAMP).*

Tipo de dado abstrato

“ [...] A abstração é nossa mais importante ferramenta mental para lidar com a complexidade. Portanto, um problema complexo não poderia ser visto imediatamente em termos de instruções de computador [...] mas, antes, em termos de entidades naturais ao próprio problema, abstraído de maneira adequada. ”

NIKLAUS WIRTH (1989)

O significado da palavra *abstrair*, segundo o Dicionário Aurélio, é considerar isoladamente um ou mais elementos de um todo. Assim, em programação, abstrair é imaginar um problema maior dividido em problemas menores, para resolvê-los isoladamente e, posteriormente, uni-los, produzindo a solução do problema.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- definir e usar tipos de dados primitivos e não primitivos;
- utilizar as estruturas de dados como tipos de dados abstratos;
- saber que trabalhar com abstração dos tipos de dados auxilia na compreensão do paradigma de programação orientada a objetos, sem entrar nos conceitos desse paradigma.



Para começar

O computador é considerado uma máquina abstrata, pois, independentemente de saber *como* ele executa as tarefas, interessa saber *o que* ele pode executar.



A abstração, em computação, é um conceito antigo, usado inicialmente para processos. SIMULA 67 foi a primeira linguagem com recursos para a abstração de dados. [Barbara Liskov e Stephen Zilles \(1974\)](#) definiram o conceito de *tipo de dado abstrato* como uma classe de objetos abstratos, afirmando: “... o que desejamos de uma abstração é um mecanismo que permita a expressão de detalhes relevantes e a supressão de detalhes irrelevantes”. Eles destacaram, ainda, que *abstração* é um termo usado em vários segmentos e por várias pessoas como a chave para a construção de um bom projeto.

Na engenharia de software, ao se adotar a divisão do problema em módulos, se estabelecem níveis de abstração. No mais alto nível de abstração, os termos usados para a declaração são os do próprio ambiente do problema. O mais baixo nível de abstração está mais próximo dos detalhes de implementação.

A divisão de um problema maior em módulos não deve ser aleatória, e a arquitetura do software gerada pode variar de acordo com a diversidade de tendências, estilos e experiência de quem o projeta. Entretanto, há técnicas que auxiliam na construção dessa divisão, de forma que o resultado final

contemple requisitos de um bom projeto, tais como a busca de módulos com alta coesão e baixo acoplamento.

A coesão e o acoplamento entre os módulos divididos afetam o nível de abstração de um projeto. A abstração é a arte de expor seletivamente a funcionalidade de interfaces coesivas e, ao mesmo tempo, esconder as implementações acopladas.

Dominar a habilidade de pensar de forma abstrata sobre os componentes de um programa de computador é um problema para o estudante de séries iniciais de cursos de computação. Enquanto não desenvolve essa habilidade, o estudante tende a ver um programa como uma coleção não estruturada de declarações e expressões. Com isso, a complexidade de um programa aumenta com o tamanho do problema a ser desenvolvido. Ao desenvolvê-la, ele tende a ver o programa como uma coleção de funções e classes. Torna-se, então, mais fácil fazer a manutenção desse programa e reutilizá-lo.

Assim, estruturas de dados, como listas lineares, pilhas e filas, devido ao conjunto de operações que definem claramente a ação sobre os dados que elas possuem, representam elementos adequados para a introdução do conceito de abstração.

Parece complicado, mas não é. Vamos lá!



Conhecendo a teoria para programar

De modo geral, todos os programas (*softwares*) podem ser denominados “processadores de informação”. São construídos para processar as informações que recebem através das entradas, aplicar-lhes transformações e produzir as saídas.

A elaboração de todo programa, respeitadas as devidas proporções, deveria seguir obrigatoriamente três fases genéricas:

- definição;
- desenvolvimento; e
- manutenção.

A *fase de definição* do *software* é aquela em que se deve determinar claramente o que se pretende atender com esse programa. Para tanto, devem-se identificar os principais requisitos do *software*, tais como:

- funcionalidade;
- desempenho desejado;
- informações a serem processadas;

- critérios de validação;
- requisitos de interface; e
- restrições do projeto.

A *fase de desenvolvimento* é aquela na qual se deve definir como será a arquitetura do *software*, como serão implementados os detalhes de procedimentos, como os dados serão traduzidos para uma linguagem de programação e, por fim, como serão os testes.

A *fase de manutenção* é a caracterizada pelas mudanças. É nessa fase que podem ser feitas mudanças para solucionar erros encontrados pelos testadores, além de adaptações e/ou melhorias.

Apesar da importância dessas três fases, entre elas destaca-se a fase de desenvolvimento, que compreende o estudo da arquitetura do *software*. A arquitetura refere-se a duas importantes características de um programa:

- a estrutura hierárquica dos componentes do *software* (módulos);
- a estrutura dos dados.

Modularidade

É um conceito bastante antigo que consiste em dividir um *software* em componentes individuais, rotulados e endereçáveis, chamados *módulos*, que, uma vez integrados, atendem aos requisitos do problema (“dividir para conquistar”). A [Figura 1.1](#) representa essa ideia.

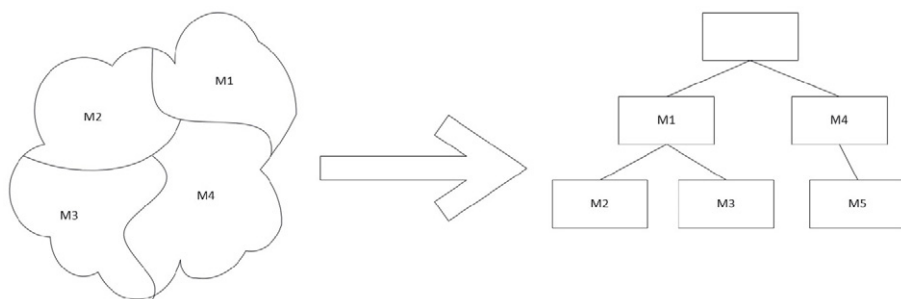


FIGURA 1.1 – Divisão do problema em módulos.

A solução de qualquer problema pode ser dada de várias formas. Essas formas são caracterizadas como diferentes níveis de abstração.

A técnica que divide o problema em módulos cria uma estrutura hierárquica de componentes de *software*. Esses componentes (módulos) podem

ser desenvolvidos utilizando-se técnicas diferentes, isto é, cada módulo, que pode ser ou não desenvolvido por meio de um único subprograma, pode ter desenvolvimento linear ou recursivo.

A definição das estruturas de dados para um software é muito importante, pois influencia todo o processo de desenvolvimento do programa.

A organização e a complexidade de uma estrutura de dados são limitadas apenas pela criatividade de quem as projeta. Entretanto, há um conjunto limitado de estruturas clássicas de dados a partir das quais é possível criar outras mais sofisticadas.

As *estruturas de dados* diferem umas das outras pela forma como as informações estão dispostas e pela forma como essas informações são manipuladas. Portanto, determinada estrutura é caracterizada pela *forma de armazenamento* dos dados e pelas *operações definidas* sobre ela. Assim, não se pode desvincular uma estrutura de dados de seus aspectos algorítmicos. A escolha de uma estrutura depende diretamente do conhecimento algorítmico das operações que a manipulam.

Algoritmos

Algoritmo é um conjunto finito de regras de definição, atribuição e controle cuja sequência lógica deve atender aos requisitos de um problema. Essas regras são executadas por uma ferramenta que as entenda, no caso o computador.

Quase sempre existe mais de uma forma de resolver um mesmo problema. Os elementos básicos que tornam um algoritmo diferente de outro são, basicamente:

- Componentes que formam o corpo do algoritmo:
 - preferência por determinadas instruções (estilo de programação);
 - divisão do programa em subprogramas;
 - recursividade ou não.
- escolha e definição das estruturas de dados.

Na solução de um problema, é necessário escolher uma abstração da realidade, isto é, definir um conjunto de dados que represente a situação. Essa escolha deve atender ao problema e estar de acordo com os recursos que a linguagem e a máquina que serão utilizadas podem oferecer.

A escolha da representação dos dados, na maioria das vezes, é guiada pelas operações que devem ser executadas sobre os dados.

Essas representações de dados são classificadas de acordo com características importantes, que dizem respeito ao conjunto de valores que uma

informação pode assumir e às operações que podem ser executadas sobre elas. Essa classificação caracteriza os *tipos de dados*.

Um tipo de dado determina o conjunto de valores a que uma constante pertence, ou que pode ser assumido por uma variável ou uma expressão, ou que pode ser gerado por uma operação ou uma função.

Os tipos de dados podem ser classificados em *primitivos* e *não primitivos* (criados a partir dos primitivos).

Tipos de dados

Cada nome (identificador) em um programa, em qualquer linguagem, tem um tipo associado a ele. Esse tipo determina que operações podem ser aplicadas ao nome e como elas devem ser interpretadas.

Tipos de dados primitivos

Também chamados de *estruturas de dados primitivas*, tipos de dados primitivos são aqueles que estão disponíveis na maioria dos computadores, como:

- Inteiros:
 - conjunto de valores: negativos, zero, positivos;
 - operações: +, -, *, /;
 - comparações: =, >, <, ≥, ≤, ≠.
- Reais:
 - conjunto de valores: parte inteira e fracionária;
 - operações: +, -, *, /;
 - comparações =, >, <, ≥, ≤, ≠.
- Lógicos:
 - conjunto de valores: verdadeiro e falso;
 - operações: e ($\wedge$), ou ($\vee$), não ($\sim$);
 - comparações: = e ≠.
- Caracteres:
 - conjunto de valores: letras, números, símbolos especiais;
 - operações: não há;
 - comparações: =, >, <, ≥, ≤, ≠.

Tipos de dados não primitivos

Tipos de dados não primitivos são aqueles construídos a partir dos tipos primitivos.

Os principais tipos são:

- arranjos (*array*) - vetores e matrizes;
- cadeia de caracteres (*strings*);
- tabelas;
- listas ligadas;
- listas lineares; e
- listas não lineares.

Tipo de dado abstrato (TDA)

É um tipo de dado que agrega os seguintes componentes:

- definição abstrata dos dados;
- ações abstratas.

Para se considerar abstrato o tipo de dado, devemos aplicar as operações definidas para ele independentemente de saber como foram implementadas ou como os dados foram armazenados, isto é, se eles foram armazenados usando alocação estática ou dinâmica de memória. O importante é saber *o que* as operações fazem, quais seus parâmetros de entrada e o que produzem como resultado, sem necessariamente saber *como* elas foram construídas.

Com base nesse conceito, vamos trabalhar alguns exemplos de arranjos (*arrays*) e cadeias de caracteres (*strings*) como *tipos de dados abstratos*. É uma forma um pouco diferente de usar esses tipos de dados, se comparada com a que se aprende num primeiro contato com linguagens de programação.

Arranjo (*array*) como tipo de dado abstrato

As operações definidas para os arranjos, em se tratando de vetores e matrizes, são as já conhecidas para esse tipo na representação matemática.

A seguir damos um exemplo de como podemos trabalhar um vetor como tipo de dado abstrato. Vale lembrar que não pretendemos, aqui, entrar em detalhes, nem usar recursos definidos para a programação orientada a objetos. Vamos apenas direcionar o pensamento para construções abstratas de dados. Posteriormente, o arranjo será utilizado em outras representações internas.

Definição do tipo como TDA – Vetor

- *Nome do tipo:* **Vetor** (pode ser de inteiros ou reais);
- *Componentes:* número de elementos e espaço de armazenamento.

Especificação das operações (alguns exemplos)

Considere:

- x, y do tipo **Vetor**;
 - k, n do tipo **Inteiro**;
 - a do tipo **Real**.
1. `produto_por_escalar ( x, k ) y`
 2. `soma_dos_elementos ( x ) a`
 3. `num_elementos( x ) n`
 4. `leitura ( ) x`
 5. `imprime ( x )`

Considerando-se a existência das operações definidas, vamos construir programas em C que implementem os seguintes problemas:

- **Problema 1-** Ler um conjunto de notas de alunos, calcular a média da turma, imprimir as notas lidas e a média da turma.
- **Problema 2-** Ler dois conjuntos de pontuações relativas aos resultados obtidos por dois participantes do Rally Paris-Dakar. Calcular a média de cada um e indicar qual deles teve a melhor média. Imprimir as pontuações lidas, as médias e quem obteve a melhor média.

Problema 1

Código 1.1

```
//definição do tipo
typedef float Vetor;
void main( )
{
    Vetor Notas;
    int NumElementos;
    Notas = leitura();
    NumElementos = num_elementos(Notas);
    float media = soma_dos_elementos (Notas)/NumElementos;
    imprime(Notas);
    printf("\nMedia: %f",media);
}
```

Problema 2

Código 1.2

```
//definição do tipo
typedef float Vetor;
void main()
{
    Vetor Participante1, Participante2;
    float Media_01, Media_02;
    int NumParticipantes1, NumParticipantes2;
    Participante1 = leitura();
    Participante2 = leitura();
    NumParticipantes1 = num_elementos(Participantes1);
    NumParticipantes2 = num_elementos(Participantes2);
    Media_01 = soma_dos_elementos(Participante1)/NumParticipante1;
    Media_02 = soma_dos_elementos(Participante2)/NumParticipante2;
    if( Media_01 > Media_02)
        printf("Media do primeiro e' Maior: %f",Media_01);
    else
    {
        if(Media_01 < Media_02)
            printf("Media do segundo e' Maior: %f", Media_02);
        else printf("\nMedias iguais : %f", Media_01);
    }
}
```

Não precisamos nos preocupar com o tamanho dos vetores nem com a forma como seus valores foram lidos, nem mesmo onde foram armazenados. Partimos do princípio de que isso já estava definido e apenas usamos as operações. Isso faz com que passemos a nos preocupar com a solução do problema, e não com o local onde vamos armazenar os dados (local de memória, dinâmica ou estática).

Se executarmos esses exemplos, certamente o compilador não reconhecerá o tipo de dado – *Vetor* –, nem as operações – *leitura*, *soma_dos_elementos*, *num_elementos* e *imprime* –, pois fomos nós que as criamos. Para isso, teríamos de criar, em algum momento, um arquivo especial que compusesse nossa biblioteca de operações, como as que o compilador possui.

Em C/C++, esse arquivo é chamado de *header*, e, em nosso exemplo, pode ser chamado de *Vetor.h*. Ele conterá a definição do tipo e as operações construídas.

Para compreender melhor o tipo de dado abstrato, vamos utilizar como exemplo a cadeia de caracteres.

Cadeia de caracteres como TDA

A estrutura cadeia de caracteres (*string*) é um arranjo de caracteres que possui tamanho variável. Cada linguagem de programação possui uma forma própria de manipular cadeias de caracteres.

A cadeia de caracteres, na forma como está implementada nas linguagens de programação, representa um tipo de dado abstrato. Vejamos:

- ela possui um campo de definição: *char nome[]*; e
- um conjunto de operações definidas sobre essa definição.

Observe alguns exemplos de operações na linguagem C/C++:

```
strcpy(cc1, cc2); // faz uma cópia de cc2 em cc1
strcmp(cc1, cc2); // compara as duas cadeias de caracteres
strlen(cc1); // determina o comprimento de cc1
strcat(cc1, cc2); // concatena as duas cadeias de caracteres
```

Elas podem ser usadas adicionando-se ao programa a biblioteca *string.h*.

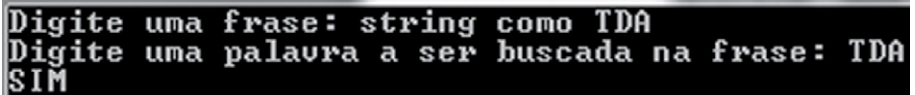
Essas e outras operações para manipulação de cadeias de caracteres existentes na biblioteca *string.h* podem ser encontradas em livros de programação em C/C++ ou no *help* do próprio compilador. Além disso, de acordo com o problema, outras operações podem ser criadas.

Vejamos o problema a seguir, que utiliza uma cadeia de caracteres: suponha que uma cadeia de caracteres *S* contenha uma frase qualquer. Vamos construir um programa que, utilizando as operações disponíveis na biblioteca *string.h*, verifique se a frase possui pelo menos uma ocorrência de um caractere *P*, também lido. Em caso afirmativo, imprimir SIM; caso contrário, imprimir NAO.

Código 1.3

```
#include <stdio.h> //biblioteca com as operações de entrada e saída
#include <string.h> //biblioteca com as operações de cadeia de caracteres
#define TamFrase 20 //tamanho máximo da frase a ser lida: 19 caracteres
#define TamPalavra 10 //tamanho máximo da palavra: 9 caracteres
void main()
{
    char S[TamFrase];
    char P[TamPalavra];
    printf("Digite uma frase: ");
    gets(S); //leitura da frase
    printf("Digite uma palavra a ser buscada na frase: ");
    scanf("%s", P);
    if( strstr(S,P))
        printf("SIM");
    else
        printf("NAO");
}
```

Resultado esperado (Figura 1.2):



```
Digite uma frase: string como TDA
Digite uma palavra a ser buscada na frase: TDA
SIM
```

FIGURA 1.2: Resultado da execução do código apresentado.

Atenção

Lembre-se de que, na declaração `char X[10]`, por exemplo, de uma variável do tipo cadeia de caracteres, no máximo 9 caracteres serão armazenados em X. A última posição será ocupada com o caractere de fim de cadeia de caracteres, representado por: `'\0'`.



Criando TDA específico

Em várias linguagens, é possível criar tipos de dados abstratos específicos para controlar e armazenar informações necessárias para seus desenvolvimentos. Os TDAs podem ser compostos por tipos primitivos e por outros TDAs, algumas vezes servindo de referência para o próprio TDA.

Imagine que um TDA seja uma grande região de memória subdividida em pedacinhos. Esses pedacinhos são os espaços suficientes para guardar o tipo primitivo que compõe esse TDA.

Fazendo uma analogia com o mundo real, um TDA seria um quarteirão de nossas cidades. Sabemos que os quarteirões são subdivididos em terrenos de diferentes tamanhos. Nesse caso, os terrenos representam as regiões de memória alocadas para um tipo primitivo dentro do TDA.

Em linguagem C, os TDAs são criados por meio da definição de *structs*, que pode ser composta por tipos primitivos e/ou por outras *structs*. Em geral, um TDA é definido como um conjunto de informações que devem seguir juntas para o bom funcionamento de um programa, como um programa que controlaria uma secretaria de faculdade, por exemplo. Nesse caso, as informações básicas, para eles, são as relacionadas aos alunos, como seu registro acadêmico, nome completo, e-mail, telefone e outros dados. Veja a seguir um exemplo, em linguagem C, que cria um TDA com essas informações:

Código 1.4

```
struct aluno
{
    char nome[50];
    int registro_academico;
    char email[200];
    char telefone[10];
    int ano_egresso;
};
```



Vamos programar

Vejamos agora como seriam as representações dos tipos *Vetor* e *Cadeia de caracteres*, estudados na seção anterior, tanto em Java como em Python.

Java

Código 1.1

```
import java.util.*;
public class ProgramaTipoVetor
{
    public static void main(String[] args)
    {
        Vetor Notas;
        int NumElementos;
        float media;
        Notas = leitura();
        NumElementos = num_elementos(Notas);
        media = soma_dos_elementos(Notas)/NumElementos;
        imprime(Notas);
        System.out.print("\nMedia:"+media);
    }
}
```

Código 1.2

```
import java.util.*;
public class ProgramaTipoVetor
{
    public static void main(String[] args)
    {
        float[] Participante1 = new float[10];
        float[] Participante2 = new float[10];
        float Media_01, Media_02;
        int NumParticipantes1, NumParticipantes2;
        Participante1 = leitura();
        Participante2 = leitura();
        NumParticipantes1 = num_elementos(Participantes1);
        NumParticipantes2 = num_elementos(Participantes2);
        Media_01= soma_dos_elementos(Participante1)/NumParticipante1;
        Media_02 = soma_dos_elementos(Participante2)/NumParticipante2;
        if( Media_01 > Media_02)
            System.out.println("Media do primeiro eh Maior: " + Media_01);
        else
        {
            if( Media_01 < Media_02)
                System.out.println("Media do segundo eh Maior: " + Media_02);
            else
                System.out.println("Medias iguais : " + Media_01);
        }
    }
}
```

**Dica**

Em Java, o operador + é utilizado para “somar” uma cadeia de caracteres (*string*) com outra. Isso significa que vai concatenar uma cadeia de caracteres com outra.

Código 1.3

```
import javax.swing.JOptionPane;
public class TipoString
{
    public static void main(String[] args)
    {
        String S;
        String P;
        String Saida = "";
        //leitura da frase
        S = JOptionPane.showInputDialog("Digite uma frase: ");
        //leitura da palavra
        P = JOptionPane.showInputDialog("Digite uma palavra: ");
        if( S.indexOf(P) != -1)
            Saida += "SIM";
        else
            Saida += "NAO";
        JOptionPane.showMessageDialog(null, Saida, "A palavra - "+P+" - existe?",
JOptionPane.PLAIN_MESSAGE);
    }
}
```

Phyton**Código 1.1**

```
Notas = []
Notas = leitura()
NumElementos = len(Notas)
media = float(sum(Notas))/NumElementos
print Notas
print "Media: ", media
```

Código 1.2

```
Participantes1 = leitura()
Participantes2 = leitura()
NumParticipantes1 = len(Participantes1)
NumParticipantes2 = len(Participantes2)
Media_01 = float(sum(Participantes1)/NumParticipantes1)
Media_02 = float(sum(Participantes2)/NumParticipantes2)
if Media_01 > Media_02:
    print "Media do primeiro e' Maior: %f" %(Media_01)
elif Media_01 < Media_02:
    print "Media do segundo e' Maior: %f" % (Media_02)
else:
    print "Medias iguais : %f" % (Media_01)
```

Código 1.3

```
Saida=""
S = raw_input("Digite uma frase: ")
P = raw_input("Digite uma palavra: ")
if P in S:
    Saida += "SIM"
else:
    Saida += "NAO"
print "A palavra -", P, "- existe?\n", Saida
```



Para fixar

Vamos praticar definindo outros tipos de dados que não estão disponíveis nas bibliotecas das linguagens, mas que podem ser úteis em determinadas aplicações, como em projetos de ensino de matemática, por exemplo. Vamos criar o *tipo de dado natural*.

Na matemática, os naturais são os números inteiros, sem os negativos. Lembre-se de que nas operações de subtração com os números naturais há uma restrição: o minuendo deve ser sempre maior ou igual ao subtraendo.

Criação do tipo de dado natural

- Definição do tipo:

```
typedef int NATURAL;
```

- Especificação das operações:

```
Zero ( )→0;  
Adição ( natural, natural )→natural  
Subtr ( natural, natural )→natural  
Mult ( natural, natural )→natural  
Divi ( natural, natural )→natural  
Igual ( natural, natural ) lógico  
Sucessor ( natural )→natural  
e outras: Maior, Menor, ...
```

- Regras: para todo $x, y \in \text{natural}$ seja:

```
Adição (Zero ( ), y )→y  
Subtr ( x, y )→z  $\in \text{Natural}$  se  $x \geq y$   
Adição ( Sucessor( x ), y )→Sucessor ( Adição ( x, y ) )
```

Agora, com base nas regras criadas, implemente as operações.
Vamos lá, você consegue!



Para saber mais

Como já mencionado, a abstração é empregada mais frequentemente em programação, que utiliza o paradigma de orientação a objetos. Em se tratando do aprendizado de estruturas de dados como tipo de dado abstrato, o livro *Estruturas de dados usando C* (Tenenbaum e Langsam, 2004) traz uma aplicação de um dicionário como um tipo de dado abstrato, que, além de utilizar o conceito de abstração como apresentado neste capítulo, integra técnicas de pesquisa que você aprenderá mais adiante neste livro. Vale a pena conferir para consolidar seu aprendizado.

Na construção dos módulos de um programa, é necessário observar dois requisitos importantes para alcançar a independência desejada: a coesão e o acoplamento entre os módulos. Esses requisitos devem atender às regras de alta coesão e baixo acoplamento. O livro *Engenharia de software: uma abordagem profissional* (Pressman, 2011) apresenta uma abordagem bastante completa sobre as técnicas de desenvolvimento de *software*. Boa leitura!



Navegar é preciso

Existem muitos *sites* de docentes de universidades que tratam do assunto *tipo de dados abstratos*, também tratado como *tipo abstrato de dados*.

No Portal Universia (<http://mit.universia.com.br/>), há uma série de conteúdos dos cursos do MIT, traduzidos para o português. Veja a aula sobre tipos abstratos em: http://mit.universia.com.br/6/6.170/pdf/6.170_lecture-05.pdf. Vale a pena conferir a abordagem desse assunto e outros de seu interesse.

Exercícios

1. Defina o TDA, **FRAÇÃO**. Utilize dois números inteiros, um representando o numerador e outro o denominador.
2. Defina o TDA, **COMPLEXOS**. Utilize dois números reais, x e y , representando $x + yi$.
3. Defina o TDA, **MATRIZ**. Crie operações básicas conhecidas, como soma de duas matrizes, produto de duas matrizes, cálculo da matriz inversa, etc.
4. Em linguagem C, declare uma **struct** que contemple as seguintes informações: nome_do_doador, tipo_sanguineo, RG, CPF e Qtd_vezes_doou.

Glossário

Coesão: grau de dependência de um módulo com relação aos demais. Um módulo coeso deve restringir-se a desenvolver uma tarefa específica.

Acoplamento: medida de relacionamento entre os módulos.

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- HOROWITZ, E.; SAHNI, S. *Fundamentos de estruturas de dados*. Rio de Janeiro: Campus, 1987.

- LISKOV, B.; GUTTAG, J. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Boston: Addison-Wesley, 2000.
- LISKOV, B.; ZILLES, S. *Programming with Abstract Data Types*. ACM SIGPLAN Notices, 9(4):50-9, 1974.
- PIVA, D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Elsevier, 2012.
- PRESSMAN, R. S. *Engenharia de software: uma abordagem profissional*. 7ª ed. Porto Alegre: McGraw-Hill-Artmed, 2011.
- SCHMIDT, B. *The Learning C/C++ - Driving you to Abstraction*. *C/C++ Users Journal*, 15(1): Article 8, 1997.
- TENENBAUM, A. M.; LANGSAM, Y.; AUGENNSTEIN, M. J. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Depois de aprendermos o que são os tipos de dados abstratos e como utilizá-los, vamos empregá-los em estruturas de dados. Mas, antes de começar, teremos de aprender outros conceitos que serão necessários para trabalhar com as pilhas, filas, listas lineares e árvores.

Vamos lá!

Mapa de memória de um processo

“Existem apenas dois tipos de planos de batalha, os bons e os maus. Os bons falham, quase sempre, devido a circunstâncias imprevistas que fazem, muitas vezes, que os maus sejam bem-sucedidos.”

NAPOLEÃO BONAPARTE

Conhecer o inimigo e o campo de batalha, geralmente, são pré-requisitos da vitória.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer e identificar as principais áreas da memória;
- conhecer as implicações do gerenciamento eficiente da memória e seu impacto no desempenho do sistema;
- saber em que parte da memória as informações e, consequentemente, as estruturas de dados são armazenadas;
- alocar e liberar blocos de dados alocados dinamicamente na memória.



Para começar

Numa batalha, a movimentação dos soldados e seus armamentos pode ser fator decisivo para a vitória.



Conhecer o campo onde a batalha ocorrerá, facilita muito essa movimentação e deslocamento dos soldados e de seus armamentos.

No caso dos sistemas computacionais, a programação e a manipulação de estruturas de dados podem ser vistas como uma batalha entre o sistema operacional e a grande quantidade de programas que estão em simultânea execução, incluindo aquele que você acabou de desenvolver. Todos batalham para obter o máximo de memória possível e terminar suas tarefas no mais curto espaço de tempo. De outro lado, o sistema operacional tenta colocar uma ordem em tudo isso, limitando alguns acessos e alocações.

Conhecer esse “terreno”, isto é, saber onde podem ou não ser alocadas determinadas informações pode melhorar muito o desempenho de seu programa, possibilitando a otimização de todo o sistema.

Neste capítulo, aprenderemos sobre o processo de alocação da memória, como ela é utilizada, quais são os espaços reservados para ela e a forma correta de a utilizarmos. Vamos lá!

Atenção

Conhecer os processos de alocação dinâmica e linear que ocorrem na memória possibilita a construção de programas que utilizam de forma mais inteligente as áreas da memória, consequentemente, sua execução ocorre de maneira mais otimizada, melhorando o desempenho global do programa.



Não sei se isso ocorre com você, mas, quando inicio meu trabalho, a minha mesa está limpinha. No fim do dia, eu não consigo encontrar com facilidade os documentos ou objetos que procuro. Em geral a mesa fica bem bagunçada.

Em sua opinião, qual(is) procedimento(s) poderia(m) ser realizados ao longo do dia para que essa situação não chegasse ao extremo?



Dica

Para responder a essa questão, pense que a mesa representa um espaço limitado de trabalho. No início, está vazia. Depois de um dia de trabalho intenso, muitos objetos, como *notebook*, grampeador, uma porção de folhas de papel, livros, entre muitas outras coisas, ocupam esse espaço.

E então? Em quais soluções você pensou?

Talvez tenha pensado em dividir o espaço da mesa em áreas, e, em cada uma dessas áreas, dispor certos objetos, como da seguinte forma: área das folhas de papel, área do *notebook*, área dos livros, área das ferramentas (grampeador, extrator, etc.).

Essa é uma boa solução.

Também poderíamos colocar sobre a mesa apenas as coisas necessárias para realizar determinada tarefa. Quando essa tarefa fosse concluída, os objetos seriam, então, retirados da mesa, abrindo espaço para a tarefa seguinte. Isso otimizaria o processo e, no fim do dia, a mesa não estaria uma bagunça!

Transferindo isso para os sistemas computacionais, a mesa representaria a memória do computador, que é limitada e é o local onde colocamos os objetos necessários para a realização da maioria de nossas tarefas. A organização é fundamental para a otimização de sua utilização.

Mas como isso pode ser feito na memória? É o que vamos ver agora. Preparado? Então, vamos em frente!



Conhecendo a teoria para programar

A maioria das linguagens de programação utiliza um esquema de segmentação de memória em pelo menos quatro partes ou regiões logicamente distintas: *instruções ou o código do programa*, *variáveis globais*, *pilha* e *heap*. Cada uma dessas regiões tem funções específicas, como mostra o esquema representado na [Figura 2.1](#). Como você já deve ter deduzido, esse esquema

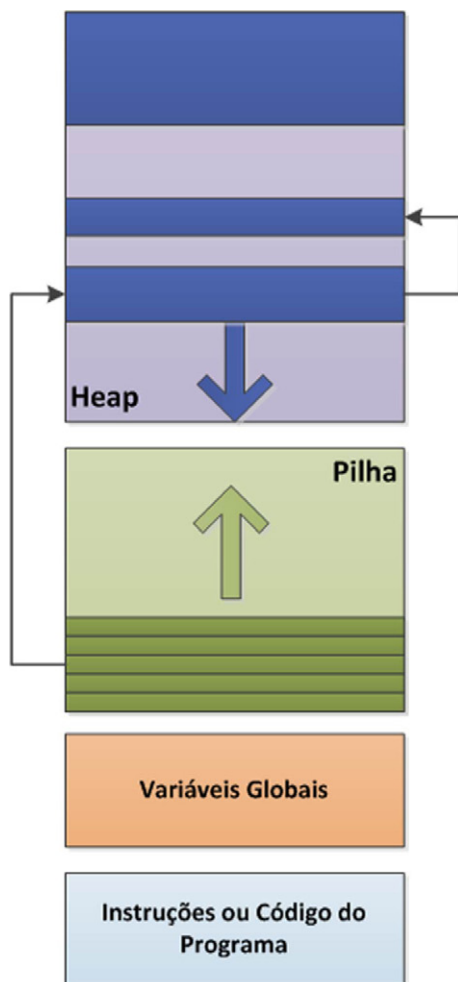


FIGURA 2.1: Esquema lógico de organização da memória nas principais linguagens de programação que utilizam alocação dinâmica.

é apenas um modelo genérico, não representando qualquer implementação real/física.

A alocação estática de memória utiliza, fundamentalmente, a região das variáveis globais. Nesse tipo de alocação, é preciso prever toda a memória de que aquele tipo de dados possa necessitar ao longo da execução do programa. O tamanho máximo de alocação possível nesse tipo é ditado pelo *hardware* (tamanho da memória “endereçável”).

O esquema representado na [Figura 2.1](#) permite que se tenha uma ideia dos diferentes papéis assumidos pela memória em linguagens que utilizam alocação dinâmica e linear.

O *heap* é utilizado para a alocação dinâmica dos objetos/dados; já a pilha controla a alocação linear, principalmente para procedimentos sequenciais.

Em linguagem Java, esse processo é feito automaticamente, ao passo que em C/C++ essa alocação é feita manualmente.

Pode-se observar, na [Figura 2.1](#), que o *heap* abriga alguns blocos de informações, que indicam os dados alocados dinamicamente. Olhando com atenção, percebe-se que entre um bloco e outro existe um espaço de memória não ocupado. Esse espaço é chamado de *fragmentação*, e é um dos problemas mais sérios que podem ocorrer quando utilizamos alocação dinâmica de memória, pois não pode ser utilizado, isto é, torna-se um espaço “perdido” na memória.

Na pilha, os blocos podem representar sequências de instruções. Também podem existir indicações da pilha para o *heap*, e de blocos do *heap* para outros blocos do *heap*. Isso acontece quando trabalhamos com ponteiros (listas encadeadas e árvores).

Nas linguagens de programação, essas duas regiões são locais imaginários da memória. Não interessa ao programador saber exatamente onde elas estão, nem como podem ser manipuladas. Mas até mesmo no caso das linguagens Java e Python, que contam com um processo de limpeza automática dessas áreas alocadas e não mais utilizadas (o *garbage collector*, ou coletor de lixo), é interessante conhecê-las, para “forçar” a limpeza nos casos que possam comprometer o desempenho.

Já em linguagens em que a alocação e a liberação de espaço são feitas manualmente, é fundamental ter esse conhecimento para adquirir um controle mais efetivo do processo.

Atenção

A disposição desses elementos na memória, mesmo quando utilizamos C ou C++, pode variar de compilador para compilador, e de sistema operacional para sistema operacional.



A partir de agora, vamos nos concentrar na alocação dinâmica. Não abordaremos a alocação estática por suas características e simplicidade de utilização.

Em linguagem C, existem quatro funções principais para trabalhar com a alocação dinâmica de memória:

- *Malloc* – permite que seja feita a alocação de uma nova área de memória para uma estrutura. Para tanto, o número de bytes que se deseja

alocar deve ser informado. Se existir esse espaço na memória, a função retorna um endereço de memória, que deve ser colocado em uma variável do tipo ponteiro.

Caso não seja possível alocar o espaço na memória, a função retorna NULL.



Dica

A função *malloc* retorna um ponteiro para o tipo *void*. Portanto, deve-se utilizar o *typecast* para transformar esse endereço no tipo de ponteiro desejado.

```
void *malloc(unsigned int num);
```

- *Calloc* – tem a mesma funcionalidade de *malloc*, entretanto, nesse caso devem ser fornecidos dois parâmetros: o tamanho da área (em bytes) e a quantidade de elementos a serem alocados. A diferença entre as duas funções (*malloc* e *calloc*) é que *calloc* aloca o espaço desejado e o inicializa com zeros.

```
void *calloc(unsigned int num, unsigned int size);
```

- *Realloc* – permite que uma área previamente alocada seja reutilizada, com sua ampliação ou redução. Isso acontece porque, muitas vezes, uma área precisa ser ampliada. Para tanto, deve-se indicar, como parâmetro para essa função, o ponteiro alocado por *malloc* e a indicação do novo tamanho.

```
void *realloc(void *ptr, unsigned int num);
```

- *Free* – permite que uma área previamente alocada seja liberada. Para tanto, deve-se indicar, como parâmetro a essa função, o ponteiro retornado quando fizemos a alocação.

```
void free(void *p);
```

Um exemplo de alocação de memória poderia ser:

Código 2.1

```
#include <stdlib.h> //Inclusão das duas bibliotecas básicas em C
#include <stdio.h>

char *s; //Declaração de duas variáveis globais estáticas do tipo ponteiro
int *r;

main()
{
    s = (char *) malloc(2000); //Alocação de 2000 bytes na memória e associação desse espaço com o ponteiro s
    r = (int *) malloc(40*sizeof(int)); //Alocação de 40 espaços do tipo inteiro e associação desse espaço ao ponteiro r.
}
```

Recuperando o esquema lógico mostrado na [Figura 2.1](#), teríamos o seguinte resultado na memória após a execução do segundo comando *malloc* ([Figura 2.2](#)):

Note que, depois da alocação, os espaços que antes estavam livres na memória ficaram reservados e são acessados pelos ponteiros *s* e *r*.

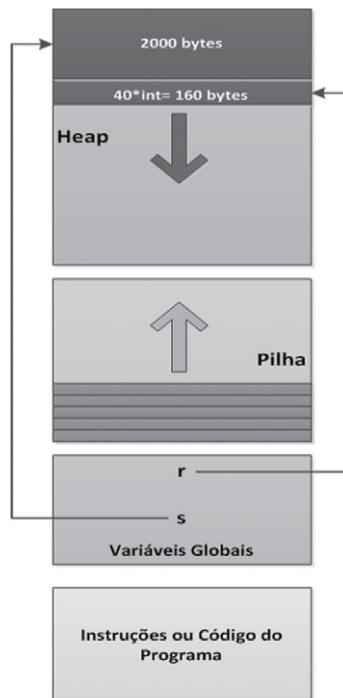


FIGURA 2.2: Resultado da execução do programa no esquema lógico de memória.

**Dica**

É muito importante que ao fazer a alocação de espaços na memória você cheque se a operação foi realizada. Para tanto, depois de um comando de alocação ou realocação de memória, verifique o ponteiro retornado. Caso ele seja NULL, isso significa que o espaço não foi alocado!

A verificação da realização bem-sucedida de uma operação de alocação pode ser feita da seguinte forma:

Código 2.2

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
    s = (char *) malloc(2000);
    if (s==NULL) {
        printf("\nErro de alocação de memória"); // Verifica se s é igual a NULL. Se essa condição for atendida,
        exit(1); // significa que o ponteiro s não recebeu o valor do endereço de
        // memória da posição inicial do espaço alocado. Portanto, não foi
        // feita a alocação.
    }
    r = (int *) malloc(40*sizeof(int));
    if (r==NULL) { // idem para o ponteiro r.
        printf("\nErro de alocação de memória");
        exit(1);
    }
}
```

Após a utilização dos espaços reservados na memória (geralmente no final do programa), é muito recomendável que eles sejam liberados. Isso é feito com a função *free*. O trecho de programa a seguir libera o espaço no fim de sua execução.

Código 2.3

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
    s = (char *) malloc(2000);
    if (s==NULL) {
        printf("\nErro de alocação de memória");
        exit(1);
    }
    r = (int *) malloc(40*sizeof(int));
    if (r==NULL) {
        printf("\nErro de alocação de memória");
        exit(1);
    }

    ...
    {outros comandos/instruções}
    ...

    free(r); // Libera o espaço alocado na memória e associado ao ponteiro r (160 bytes)

    free(s); // Libera o espaço alocado na memória e associado ao ponteiro s (2000 bytes)
}
```

A utilização da função *calloc* é semelhante à da função *malloc*. Pensando em nosso exemplo, quando fizemos a alocação de 40 elementos do tipo inteiro utilizando *malloc*, também poderíamos ter utilizado a função *calloc* da seguinte forma:

Código 2.4

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
    ...
    r = (int *) calloc(40, sizeof(int)); // Essa seria uma outra forma de fazer a alocação de 40 elementos do tipo
                                         // inteiro, utilizando a função calloc.

    if (r==NULL) {
        printf("\nErro de alocação de memória");
        exit(1);
    }

    ...
}
```

Quando precisarmos de mais espaço, poderemos utilizar a função *realloc*, que permite a alocação de mais espaço. Entretanto, na maioria das vezes, nós o fazemos para expandir o espaço necessário. Assim, em muitos casos, o lugar físico da memória terá de ser alterado. No nosso exemplo, vamos expandir a quantidade de memória associada ao ponteiro *s* para 3.000 bytes. Ficaria assim:

Código 2.5

```
#include <stdlib.h>
#include <stdio.h>
char *s;
int *r;
main()
{
    s = (char *) malloc(2000);
    if (s==NULL) {
        printf("\nErro de alocação de memória");
        exit(1);
    }
    r = (int *) malloc(40*sizeof(int));
    if (r==NULL) {
        printf("\nErro de alocação de memória");
        exit(1);
    }

    ...
    {outros comandos/instruções}
    ...

    //necessidade de expansão...

    s = (char *) realloc(3000); //Expande o espaço reservado inicialmente de 2000 para 3000 bytes.

    free(r);
    free(s);
}
```

Note que, logo depois da primeira alocação de 2.000 bytes, foi alocada outra área. Assim, não será possível expandir linearmente para 3.000 bytes no mesmo lugar. O que acontece é que um primeiro é liberado (automaticamente) e um segundo, com 3.000 bytes, é alocado.

Para melhor visualização depois do comando *realloc*, o que acontece no esquema lógico de memória apresentado na [Figura 2.2](#) é o seguinte ([Figura 2.3](#)).

A alocação dinâmica de memória e suas implicações são conceitos importantíssimos que serão tratados com mais profundidade no Capítulo 6.

Para finalizar esses conceitos básicos de alocação, falta verificar como a pilha é utilizada.

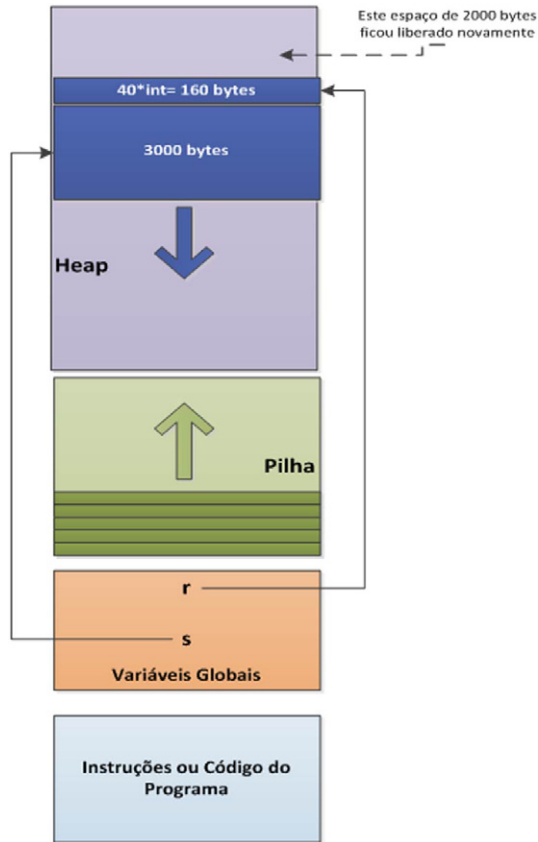


FIGURA 2.3: Resultado da execução do comando *realloc* no esquema lógico de memória.

Basicamente, a pilha organiza a ordem de chamada de funções e suas variáveis locais, estabelecendo, para cada instante da execução, a que cada identificador se refere.

Tomemos como exemplo o seguinte código em linguagem C:

Código 2.6

```
#include <stdio.h>
char *a, *b;

int funcao_X() {
    int localA, localB;
    ...
}

main() {
    a = "Global A";
    b = "Global B";
    funcao_X();
}
```

Depois da execução da segunda linha do código anterior o esquema lógico de memória poderia ser definido, conforme apresentado na [Figura 2.4](#).

Após a execução da declaração das variáveis *a* e *b*, o programa é deslocado para a função *main()*.

As duas primeiras linhas da função *main()* associam um valor constante às variáveis. Em seguida, é chamada uma nova função: *função_X()*. Ao chamá-la, as informações de chamada e as variáveis locais são empilhadas. A [Figura 2.5](#) apresenta o esquema lógico de memória após a chamada da *função_X()*.

Note que é identificado o local de onde a chamada provém. Isso serve para o controle retornar à linha subsequente à identificada — no caso, o controle retornará à linha #4 da função *main()* — quando terminar a execução da função chamada.

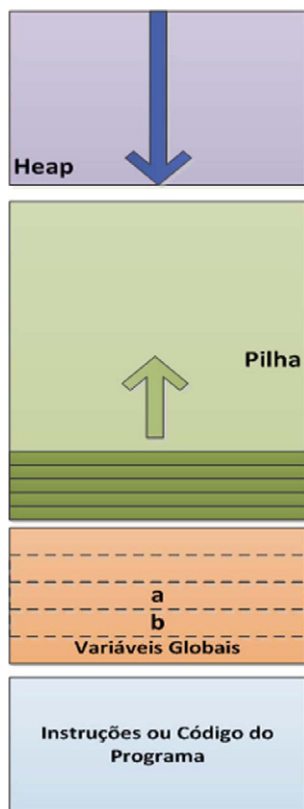


FIGURA 2.4: Resultado no esquema lógico de memória (parcial).

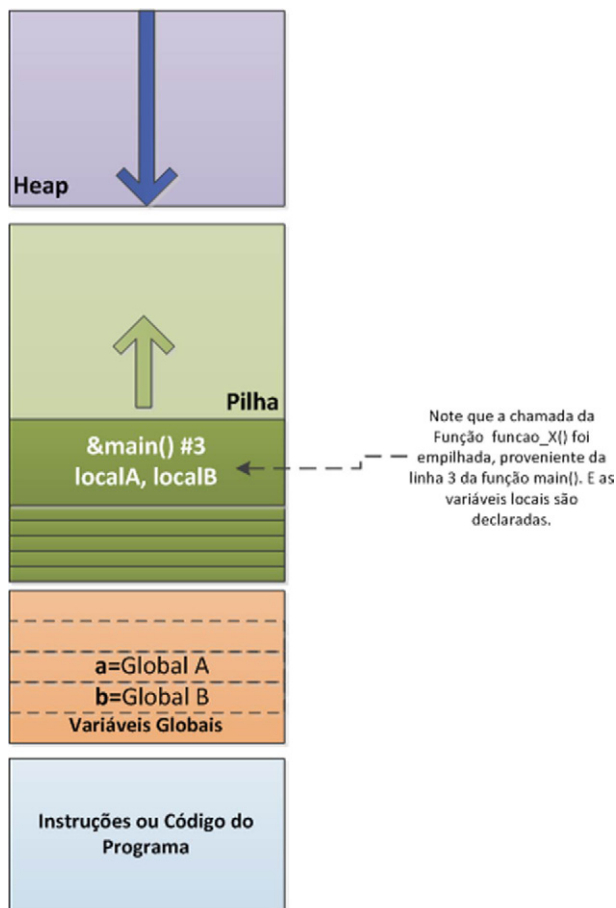


FIGURA 2.5: Resultado no esquema lógico de memória após a chamada da função `X()`.



Vamos programar

Java e Python

Diferentemente de C ou C++, programadores Java e Python não têm como gerenciar explicitamente a memória do sistema. Nessas duas linguagens, o programador desenvolve aplicações sem se preocupar com questões de alocação e liberação de memória, pois elas são realizadas automaticamente pela máquina virtual, usando para isso algoritmos específicos.

Portanto, os exemplos simples de alocação e liberação de memória apresentados neste capítulo não apresentam implementações semelhantes nas linguagens Java e Python.



Para fixar

Tomando como base o esquema lógico da memória da [Figura 2.6](#), responda às quatro questões propostas.

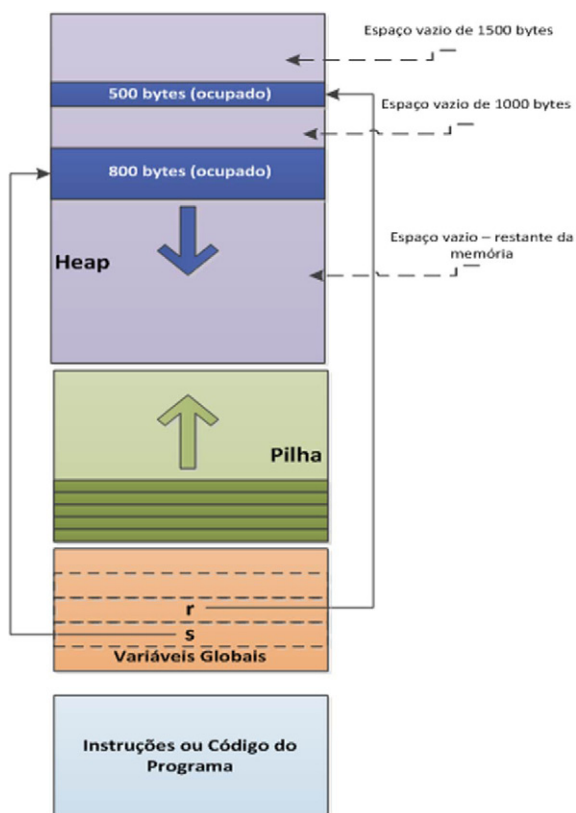


FIGURA 2.6: Estado inicial do esquema lógico da memória.

1. Tomando como base o atual estado da memória, onde seria alocado o espaço de memória alocado pelo seguinte comando:

```
t = (char *) malloc(800);
```

2. Depois que ocorreu a alocação proposta no primeiro exercício, onde seria alocada essa nova área?

```
z = (int *) calloc(100, sizeof(int));
```

3. Como ficaria o esquema lógico da memória com esse terceiro comando?

```
t = (char *) realloc(1000);
```

4. Como ficaria se o comando do exercício três fosse o representado a seguir?

```
t = (char *) malloc(2000);
```

Vamos lá, você consegue!



Papo técnico:

Existem linguagens que, mesmo possuindo o *garbage collector*, permitem ao programador interferir no gerenciamento da memória. É o caso das linguagens C++ e Java.



Para saber mais

Você poderá encontrar mais informações sobre o processo de gerenciamento da memória em livros como *Fundamental Algorithms* (KNUTH) (veja a seção *Dynamic Storage Allocation*).

Existe também um interessante documento, produzido pelo Departamento de Ciência da Computação da Universidade do Texas, em Austin, chamado *Dynamic Storage Allocation: A Survey and Critical Review*. Você pode acessá-lo no site:

<http://www.cs.northwestern.edu/~pdinda/icsclass/doc/dsa.pdf>

Lembre-se: a questão de alocação dinâmica de memória será aprofundada no Capítulo 6. No momento, você precisa apenas entender como funcionam os mecanismos de gerenciamento da memória para obter melhores resultados ao longo do curso. Bons estudos!



Navegar é preciso

Existe um site bem antigo que trata do gerenciamento de memória. Nele existe um glossário com algumas centenas de palavras, além de bons artigos e uma vasta bibliografia que você poderá consultar. Para isso, acesse o site: <http://www.memorymanagement.org/>

No site <http://www.scirp.org/journal/PaperDownload.aspx?paperID=23532>, você poderá ter acesso a um artigo que apresenta um objeto de aprendizagem específico para gerenciamento de memória. É uma boa ferramenta para entender o que acontece na memória.

Exercícios

1. Quais são as quatro principais regiões lógicas da memória? Explique cada uma delas.
2. O que é o *garbage collector*? Para que serve e onde pode ser encontrado?
3. Faça um programa que declare duas variáveis, *a* e *b*, do tipo ponteiro. Em seguida, aloque um espaço de 50 elementos do tipo inteiro para *a* e 3.000 bytes para *b*, do tipo char.
4. Tomando como base o esquema lógico da memória da [Figura 2.7](#), como ficariam as alocações propostas pelo conjunto de comandos a seguir no esquema lógico?

```
t = (char *) malloc(1800);
z = (float *) calloc(50, sizeof(float));
```

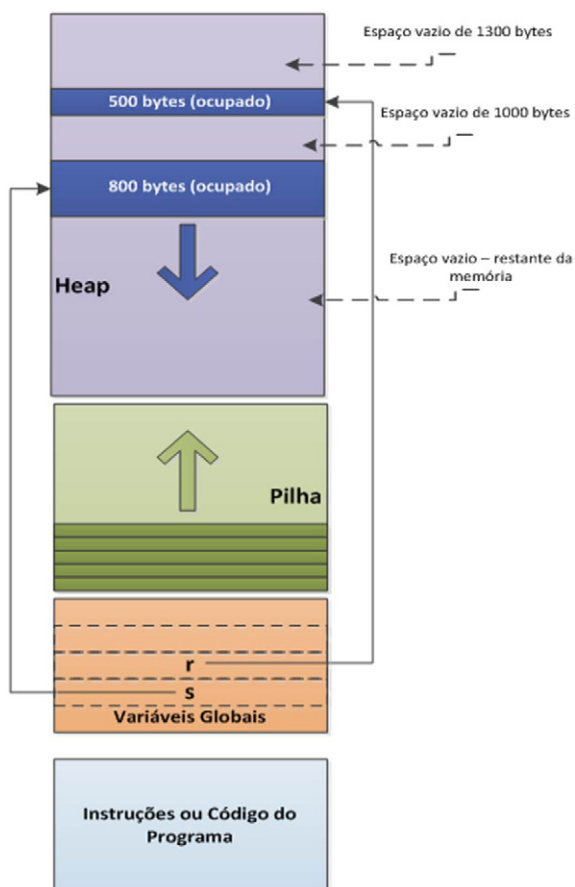


FIGURA 2.7: Estado inicial do esquema lógico da memória.

Glossário

Garbage collector: forma de gerenciamento automático da memória. Trata-se de um coletor que tenta recuperar os espaços ocupados por “lixo”, ou seja, por objetos que não são mais utilizados pelo programa. É encontrado em várias linguagens, como Java, C#, Python, entre outras.

Heap: designa uma área reservada para alocação dinâmica de objetos na memória. Existem várias formas de organização dessa área; a mais usual é uma lista encadeada de blocos livres. Essa forma é encontrada nas linguagens que permitem o gerenciamento manual de tais áreas. O processo de fragmentação dessa área pode ser um grande problema.

Referências bibliográficas

- BATES, B.; SIERRA, K. *Use a cabeça: Java*. Rio de Janeiro: Alta Books, 2007.
- CORMEN, T. H. et al. *Introduction to Algorithms: A Creative Approach*. Boston: Addison Wesley, 1989.
- SCHILDT, H. *C avançado: guia do usuário*. São Paulo: McGraw-Hill, 1989.
- SCHILDT, H. *C completo e total*. 3. e. São Paulo: Makron Books, 1996.
- SOBELMAN, G. E.; KREKELBERG, D. E. *C avançado: técnicas e aplicações*. Rio de Janeiro: Campus, 1989.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.



QUE VEM DEPOIS...

Agora que você já sabe como funciona o processo de alocação dinâmica na memória e como os dados estão distribuídos, chegou a hora de aprofundar alguns conceitos que possibilitarão a otimização de seus algoritmos. O primeiro deles é a *recursão*. Preparado? Então, bons estudos!

Recursão

“O teu êxito depende, muitas vezes, do êxito das pessoas que te rodeiam.”

BENJAMIN FRANKLIN

O êxito, na maioria dos casos, é um processo recursivo.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer e identificar objetos e processos recursivos;
- conhecer os prós e os contras da implementação recursiva e saber quando utilizar essa técnica, principalmente em termos de desempenho;
- conhecer e praticar a implementação dos principais algoritmos recursivos.



Para começar

Provavelmente, já deve ter acontecido isto com você antes: dormindo, você sonha que está dormindo e sonhando.



Em artes, esse fenômeno é conhecido como *efeito Droste* ou *miseenabyme*. Uma imagem que aparece dentro dela mesma, em um local semelhante ao da primeira imagem. Uma versão menor, que contém uma versão menor que a outra, que contém outra menor ainda, e assim por diante.

O efeito Droste tem esse nome por causa da imagem que ilustrava as caixas de cacau em pó Droste, uma das principais marcas holandesas: uma enfermeira carregando uma bandeja com uma xícara de chocolate quente e uma caixa do produto. A imagem foi feita em 1904 e mantida por décadas (Figura 3.1).



FIGURA 3.1: Caixa do cacau em pó Droste, gênese do “efeito Droste”.

Esse efeito nos remete à ideia de repetição. Mas não qualquer repetição; a repetição de um objeto dentro dele mesmo, numa ideia de *looping* ou laço contínuo. A esse processo dá-se o nome de *recursão*, ou seja, um objeto é parcialmente definido em termos dele mesmo.

A recursão pode ser encontrada na matemática, nas ciências, em computação e no cotidiano. Experimente colocar um objeto entre dois espelhos e você terá uma ideia prática da recursão infinita (ou finita, até onde sua visão lhe permitir).

A recursão está relacionada à indução matemática. E, como sabemos, a matemática é a fonte de todo o formalismo necessário em computação.

Neste capítulo, aprenderemos sobre o processo de implementação de soluções computacionais utilizando a recursividade para resolver alguns tipos de problema. Veremos também que a recursividade, mesmo sendo a forma mais prática de implementar alguns tipos de resolução, não é a técnica mais eficiente. Preparado? Vamos lá!

Atenção



A implementação de um algoritmo recursivo em uma linguagem de programação, partindo de uma definição matemática também recursiva, é praticamente direta e imediata. Por essa razão, esse tipo de algoritmo recursivo possui código muito mais legível e compacto.

O cálculo da potência p de um dado número n é obtida pela multiplicação sucessiva desse número n por ele mesmo, p vezes. Supondo que $n = 2$ e quiséssemos calcular 2^5 , teríamos: $2 \times 2 \times 2 \times 2 \times 2 = 32$. Ou seja, 5 vezes a multiplicação do número 2.

Matematicamente, tomando como a base a potência do número 2, poderíamos ter o seguinte:

$$2^n = \begin{cases} 1, & \text{se } n = 0 \\ 2 \cdot (2^{n-1}), & \text{se } n \geq 1 \end{cases}$$

Como você faria para implementar uma função em linguagem de programação para retornar o valor da *potência de 2*, dado como parâmetro um número inteiro, positivo, n ?

Dica



Para responder a essa questão, pense que existe uma condição base (inicial ou final) que corresponde a potencia ser igual a zero, resultado no valor 1. Os demais correspondem a multiplicação sucessiva do número 2, n vezes.

E então? Conseguiu?

Possivelmente, um bom programador como você, que já passou pelo tópico Algoritmos, não teria dificuldade para desenvolver uma função que utilizasse um comando de repetição interno, como o apresentado no Código 3.1, a seguir:

Código 3.1

```
...  
  
int pot2 (int n)                // função potencia de 2, recebe um valor inteiro como parâmetro.  
{  
    int k, pot=1;  
    if (n==0) return 1;        // resolve o caso base (ou situação de parada)  
    else  
        for (k=1; k<=n; k++) {  
            pot = pot * 2;      // caso recursivo (para o n-ésimo valor)  
        }  
    return pot;  
}
```

Mas, se pensarmos em termos da definição matemática, essa solução não se assemelha à definição inicial. Como isso poderia ser feito? Utilizando o conceito de *recursividade*. Ou seja, teríamos uma solução simples (ou caso base) quando o expoente fosse igual a zero. Os demais casos seriam recursivos, decrementando o valor do expoente. O Código 3.2 apresenta uma forma de resolução recursiva desse problema.

Código 3.2

```
...  
  
int pot2 (int n)                // função potencia de 2 recursiva, recebe um valor inteiro como parâmetro.  
{  
    if (n==0) return 1;        // resolve o caso base (ou situação de parada)  
    else return 2 * pot2 (n-1); // caso recursivo (para o n-ésimo valor)  
}
```

Pode ser que você ainda não tenha compreendido inteiramente o processo de recursão. Não se preocupe. É nisso que vamos nos aprofundar agora. Preparado? Então, vamos em frente!



Conhecendo a teoria para programar

Uma das mais elegantes (e, em muitos casos, complexas) técnicas da matemática é a recursividade. Ela pode ser caracterizada quando se define um objeto em termos dele mesmo.

O grande potencial da recursão está na possibilidade de poder definir elementos com base em versões mais simples desses mesmos elementos.

Em termos computacionais, trata-se de dividir um problema maior em problemas menores, em que a resolução é feita por uma mesma função (ou método), que é recorrentemente chamada. Muitos autores chamam essa técnica computacional de “dividir para conquistar”, parafraseando o grande imperador francês Napoleão Bonaparte.

Para implementar uma função (ou método) recursiva, é necessário estabelecer pelo menos dois elementos:

- uma condição de parada ou terminação. Geralmente, essa condição estabelece uma solução trivial ou um evento que encerra a autochamada consecutiva;
- uma mudança de estado a cada chamada, ou seja, o estabelecimento de alguma diferença entre o estado inicial e o próximo estado da função (ou método). Isso pode ser feito, por exemplo, decrementando um parâmetro da função recursiva.

Um exemplo clássico que se utiliza para exemplificar a recursão computacional é o cálculo do fatorial de um número n .

Apenas para aquecer, você conseguiria calcular o fatorial de um número sem utilizar a técnica de recursão? Recordando: o fatorial de um número n é igual à multiplicação dos números inteiros de 1 até n , ou seja, $1 * 2 * 3 * 4 * \dots * n$. Dessa forma, se n for igual a 4, teríamos que o fatorial de 4 é igual a 24 ou $1 * 2 * 3 * 4 = 24$.

Matematicamente, teríamos que o fatorial de um inteiro positivo n , denotado $n!$, é definido como o produto dos inteiros de 1 até n . Se $n = 0$, então $n!$ é definido como 1 por convenção. De maneira mais formal, para qualquer inteiro $n \geq 0$,

$$n! = \begin{cases} 1, & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 & \text{se } n \geq 1 \end{cases}$$

Com isso em mente, tente fazer a implementação sem utilizar a técnica de recursão. Depois, consulte o Código 3.3, a seguir, que ilustra tal possibilidade.

Código 3.3

```
...  
  
int fatorial (int n)  
{  
    int fat=1;  
    if (n==0) return fat;  
    else  
        while (n>0) {  
            fat = fat * n;  
            n--;  
        }  
    return fat;  
}
```

Conseguiu? É importante tentar!

Como você pôde ver, dependendo do valor de n (valor inteiro maior ou igual a zero), seu valor do fatorial é calculado e retornado pela função.

Como seria a implementação dessa mesma função, mas de forma recursiva? A primeira coisa a fazer é identificar o caso base ou situação de parada. No caso do cálculo do fatorial, a condição de parada é $n = 0$. Isso resulta em fatorial igual a 1. Os casos recursivos são sempre a multiplicação de um valor pelo próximo valor de n , com mudança de estado de seu valor decrementado de 1. No Código 3.4, propomos uma forma de implementar o cálculo fatorial de forma recursiva.

Código 3.4

```
...  
  
int fatorial (int n)                                // função fatorial recursiva, recebe um valor inteiro como parâmetro.  
{  
    if (n==0) return 1;                            // resolve o caso base (ou situação de parada)  
    else return n*fatorial (n-1);                  // caso recursivo (para o n-ésimo valor)  
}
```


Para melhor o entendimento do funcionamento dessa função recursiva, vamos fazer um teste de mesa ou rastreamento das recursões, supondo a chamada da função fatorial, com um parâmetro inicial igual a 4. Assim, em algum lugar, como a função *main()*, existirá uma chamada como a seguinte:

```
...  
x = fatorial(4);  
...
```

Para que você possa acompanhar com detalhes o teste de mesa, vamos numerar as linhas internas da função fatorial(). Utilizaremos o Código 3.5, a seguir:

Código 3.5

```
int fatorial (int n)                // função fatorial recursiva, recebe um valor inteiro como parâmetro.  
{  
L1   if (n==0)  
L2       return 1;                // resolve o caso base (ou situação de parada)  
L3   else return n*fatorial (n-1); // caso recursivo (para o n-ésimo valor)  
}
```

Para que a análise da execução (teste de mesa) fique mais clara, é conveniente tratar cada chamada da função fatorial() como uma nova instância da função.



Conceito

Uma instância de uma função corresponde ao processo de alocação de um novo espaço na memória (pilha) para comportar as necessidades de utilização de variáveis locais inerentes àquela função.

Na [Tabela 3.1](#), a seguir, observe os valores passados pelas chamadas e retornos.

TABELA 3.1: Teste de mesa (análise da execução) da função `fatorial()` para o valor $n=4$

Linha	Instância	Explicação	Valor n	Passagem de parâmetro	Retorno
L1	1	n não é ≤ 1	4		
L3	1	Chama <code>fatorial()</code>	4	3	
L1	2	n não é ≤ 1	3		
L3	2	Chama <code>fatorial()</code>	3	2	
L1	3	n não é ≤ 1	2		
L3	3	Chama <code>fatorial()</code>	2	1	
L1	4	n é ≤ 1	1		
L2	4	Retorna 1	1		1
L3	3	Retorna $2 \cdot 1$	2		2
L3	2	Retorna $3 \cdot 2$	3		6
L3	1	Retorna $4 \cdot 6$	4		24

Esquemáticamente, depois da primeira chamada da função `fatorial()`, teríamos a seguinte sequência de chamadas, ilustrada na [Figura 3.2](#).

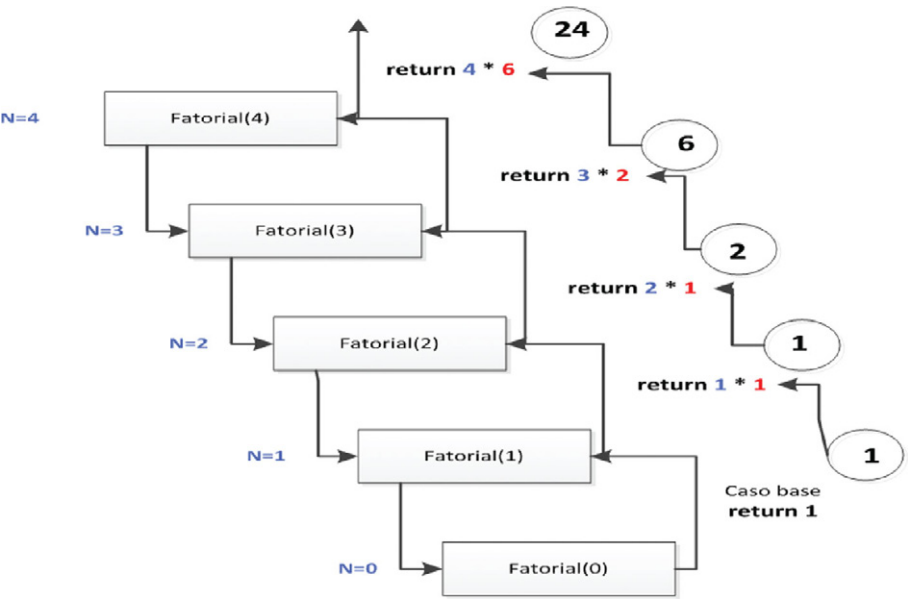


FIGURA 3.2: Sequência de chamadas da função `fatorial` (recursividade).

No Capítulo 2, mostramos como a memória é utilizada pelos processos de alocação dinâmica, pelas funções e suas variáveis. Fazendo uma breve recapitulação, no caso do cálculo do fatorial de um número n , observe que, para cada chamada da função fatorial, um novo espaço é alocado para empilhar essa nova função. Por isso, se o valor n for muito elevado, poderá ocorrer um estouro da pilha, resultando em erro de execução.

Diante disso, você pode, nesse momento, estar se perguntando: qual é a vantagem de utilizar uma função recursiva em comparação com uma não recursiva?

Embora a implementação recursiva, na maioria dos casos, seja mais simples que a versão iterativa, não existe nenhuma razão determinante para preferir a versão recursiva à iterativa. Em muitos casos, as versões recursivas consomem maior número de recursos (principalmente memória e processamento) e são muito mais difíceis de testar quando há muitas chamadas.

Entretanto, o que pode ser considerado positivo em sua utilização é a obtenção de códigos mais “enxutos” e mais fáceis de compreender, e, consequentemente, mais fáceis de implementar em linguagens de programação.



Atenção

A grande maioria dos algoritmos recursivos consome mais recursos computacionais. Por esse motivo, deve-se ter muita cautela ao utilizá-los.

Você deve utilizar a recursão quando:

1. o problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente, se comparada com a versão iterativa;
2. o algoritmo se torna compacto, sem perda de clareza ou generalidade;
3. é possível prever que o número de chamadas (e, consequentemente, a alocação na pilha) não vão provocar interrupção no processo.

Você NÃO deve utilizar a recursão quando:

1. a solução recursiva causa ineficiência, se comparada com a versão iterativa;
2. existe uma única chamada do procedimento/função recursiva no fim ou no começo da rotina, com o procedimento/função podendo ser transformado numa iteração simples;
3. o uso de recursão acarreta número maior de cálculos que a versão iterativa;

4. a recursão é de cauda;
5. parâmetros consideravelmente grandes têm que ser passados por valor;
6. não é possível prever o número de chamadas que podem causar sobrecarga na pilha.



Papo técnico

Recursão de cauda é aquela que faz a chamada dos casos recursivos (ou valores subsequentes) no final da função, após testar ou apresentar o n -ésimo valor. Ou seja, não existe processamento a ser feito depois de encerrada a chamada recursiva. O cálculo do fatorial, a sequência de Fibonacci, a soma de vetores são exemplos de recursão de cauda.

A grande vantagem da utilização de algoritmos recursivos é que, em certos casos, estes proporcionam uma forma mais simples e clara de se expressar em linguagem computacional (e, consequentemente, seu processamento). Contudo, isso não garante a melhor solução, principalmente em termos de tempo e consumo de memória.

É isso que podemos notar em outro exemplo clássico de algoritmos recursivos: o cálculo da sequência de Fibonacci. Quem leu o livro ou assistiu ao filme *Código Da Vinci*, história escrita por Dan Brown, deve se lembrar dessa sequência de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Essa sequência é composta por números naturais, e seus dois primeiros termos são 0 e 1. Cada termo subsequente corresponde à soma dos dois termos precedentes.

Matematicamente, teríamos o seguinte:

$$F(n) = \begin{cases} 0, & \text{se } n=0 \\ 1, & \text{se } n=1 \\ F(n-1)+F(n-2) & \text{se } n>1 \end{cases}$$

Note que a formulação matemática apresenta uma sequência definida recursivamente.

Portanto, pela formulação matemática, fica fácil convertê-la em uma implementação em linguagem computacional recursiva. O Código 3.6 apresenta essa função recursiva.

Código 3.6

```
...

int Fibonacci (int n) // função Fibonacci recursiva, recebe um valor inteiro como parâmetro.
{
    if ((n==0) || (n==1)) return n; // resolve os dois casos base (ou situação de parada)
    else return Fibonacci (n-1) + Fibonacci (n-2); // caso recursivo (para o n-ésimo valor)
}
```

Para a resolução do caso n é preciso que os casos $n-1$ e $n-2$ sejam, ambos, resolvidos. Isso leva a uma quantidade de chamadas (a função Fibonacci) exponencial. Para entender isso melhor, suponha que você queira calcular, pelo método recursivo, a sequência dos 7 primeiros números da sequência. Considerando que o primeiro é 0, você passaria como parâmetro o valor de $n = 6$. A [Figura 3.3](#) ilustra essa sequência de chamadas.

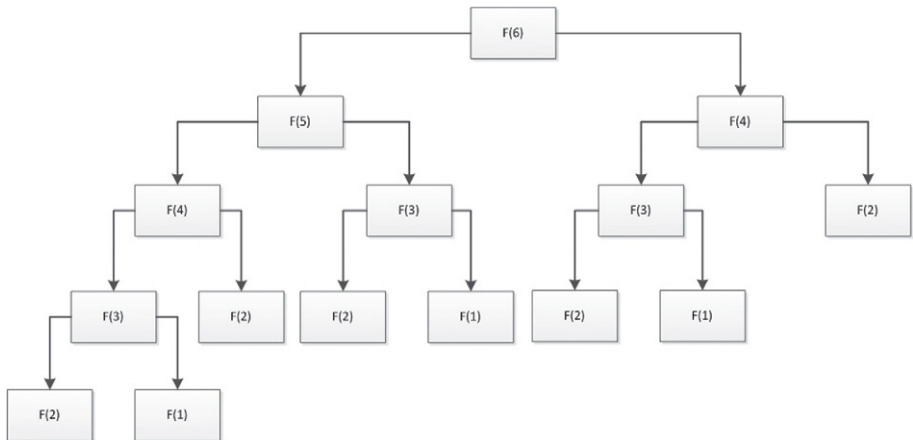


FIGURA 3.3: Sequência de chamada da função Fibonacci para $n=6$.

Note que, antes de calcular $\text{Fibonacci}(4)$, é preciso calcular $\text{Fibonacci}(3)$ e $\text{Fibonacci}(2)$, respectivamente, $n-1$ e $n-2$. Além disso, o cálculo não é reaproveitado. Por exemplo, $\text{Fibonacci}(3)$, representado na [Figura 3.2](#) por $F(3)$, é calculado três vezes, e $\text{Fibonacci}(2) - F(2)$ é calculado cinco vezes.

Em virtude dessa repetição de cálculos e chamadas exponenciais, a versão iterativa é muito mais eficiente. O Código 3.7 apresenta a versão iterativa do cálculo da sequência de Fibonacci.

Código 3.7

```
...

int Fibonacci (int n)                // função Fibonacci iterativa, recebe um valor inteiro como parâmetro.
{
    int k, fant=0, fpos=1, f;
    if ((n==0) || (n==1)) return n;  // os dois casos base: F(0)=0 e F(1)=1.
    for (k=2; k<=n; k++) {
        f=fant+fpos;
        fant=fpos;
        fpos=f;
    }
    return f;                        // retorna o valor do n número da sequência de Fibonacci
}
```

Tomando como base um computador desktop simples e comparando o tempo de execução da versão recursiva com a versão iterativa, teríamos algo semelhante ao apresentado na [Tabela 3.2](#).

TABELA 3.2: Comparativo, em termos de tempo de execução, dos algoritmos recursivos e iterativos para cálculo da sequência de Fibonacci

n	20	30	50
Recursivo	1 s	2 min	21 dias
Iterativo	1/3 ms	1/2 ms	3/4 ms

Diferentemente desses dois exemplos (fatorial e Fibonacci), a solução recursiva para o problema ou quebra-cabeça conhecido como *Torre de Hanoi* apresenta, além de maior clareza e simplicidade, um desempenho equivalente ao da solução iterativa.

O problema da Torre de Hanoi foi publicado na forma de um jogo de tabuleiros em 1883, pelo matemático Edouard Lucas (com o codinome Professor N. Claus de Siam, e consistia em transferir, com o menor número possível de movimentos, a torre composta por n discos do pino A (origem) para o pino C (destino), utilizando o pino B como auxiliar. Apenas um disco poderia ser movido por vez. Além disso, um disco não poderia ser colocado sobre outro de menor diâmetro. A [Figura 3.4](#) ilustra a situação inicial do problema.

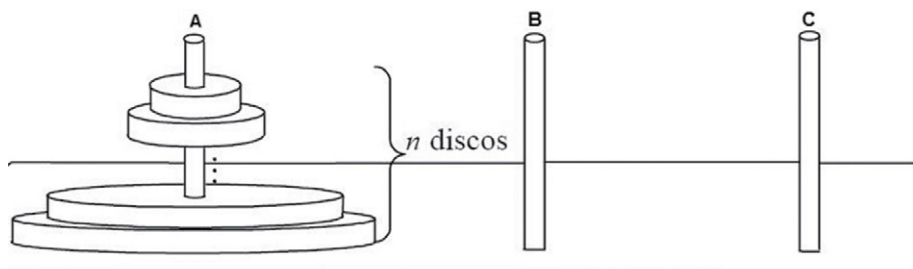


FIGURA 3.4: Situação inicial do problema da Torre de Hanoi.

A quantidade mínima de movimentos (M), dado um número n de discos, pode ser obtida pela seguinte equação matemática: $M = 2^n - 1$. Portanto, se tivermos 3 discos, o número mínimo de movimentos será 7. Se tivermos 5 discos, o número mínimo será 31, e assim por diante.

A ideia da utilização da recursão consiste em dividir um problema maior (n discos) em um problema menor ($n - 1$ discos) em cada uma das chamadas, até chegar ao problema mais simples: apenas um disco, que consiste em mover diretamente do pino A para o pino C.

Assim, se tivermos n discos, basta transferir os $n - 1$ discos de A para B, mover o maior disco de A para C e transferir os $n - 1$ discos de B para C. A cada passagem (transferir x discos), faríamos o decremento de um disco, até que restasse apenas um deles. A sequência de ilustrações na [Figura 3.5](#), representa essa ideia.

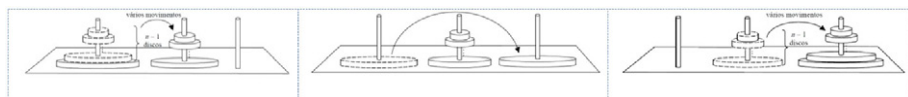


FIGURA 3.5: Ideia central para a resolução recursiva do problema da Torre de Hanoi.

O algoritmo recursivo que resolve esse problema é bem simples. Uma possível implementação é apresentada no Código 3.8, a seguir:

Código 3.8

```
...  
  
void MoveTorre (int n, char Orig, char Dest, char Aux) // função principal recursiva da Torre Hanoi  
{  
    if (n=1)  
        MoveDisco(n, Orig, Dest); // caso base ou condição de parada (tem apenas um disco).  
    else {  
        MoveTorre(n-1, Orig, Aux, Dest); // casos recursivos  
        MoveDisco(n, Orig, Dest);  
        MoveTorre(n-1, Aux, Dest, Orig);  
    }  
}  
  
void MoveDisco (int disco, char Orig, char Dest) // função que apenas mostra o movimento efetuado.  
{  
    printf("\nMovimento: Disco %i de %c --> %c", disco, Orig, Dest);  
}
```

Seria bem interessante se você tentasse resolver esse mesmo problema sem utilizar a recursão. Você vai perceber que será extremamente difícil. E, como o problema da Torre de Hanoi, existem outros contextos nos quais a utilização de recursividade traz uma resolução mais simples e elegante, com a mesma eficiência.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação JAVA e PHYTON.

Java

Código 3.1

```
public static int pot2 (int n) // função potencia de 2, recebe um valor inteiro como parâmetro.
{
    int k, pot=1;
    if (n==0) return 1; // resolve o caso base (ou situação de parada)
    else
        for (k=1; k<=n; k++) {
            pot = pot * 2; // caso recursivo (para o n-ésimo valor)
        }
    return pot;
}
```

Código 3.2

```
public static int pot2 (int n) // função potencia de 2 recursiva, recebe um valor inteiro como parâmetro.
{
    if (n==0) return 1; // resolve o caso base (ou situação de parada)
    else return 2 * pot2(n-1); // caso recursivo (para o n-ésimo valor)
}
```

Código 3.3

```
public static int fatorial (int n)
{
    int fat=1;
    if (n==0) return fat;
    else
        while (n>0) {
            fat = fat * n;
            n--;
        }
    return fat;
}
```

Código 3.4

```
public static int fatorial (int n) // função fatorial recursiva, recebe um valor inteiro como parâmetro.
{
    if (n==0) return 1;           // resolve o caso base (ou situação de parada)
    else return n*fatorial(n-1);  // caso recursivo (para o n-ésimo valor)
}
```

Código 3.5

```
public static int fatorial (int n) // função fatorial recursiva, recebe um valor inteiro como parâmetro.
{
L1    if (n==0)
L2        return 1;               // resolve o caso base (ou situação de parada)
L3    else return n*fatorial(n-1); // caso recursivo (para o n-ésimo valor)
}
```

Código 3.6

```
public static int Fibonacci (int n) // função Fibonacci recursiva, recebe um valor inteiro como parâmetro.
{
    if ((n==0) || (n==1)) return n; // resolve os dois casos base (ou situação de parada)
    else return Fibonacci(n-1) + Fibonacci(n-2); // caso recursivo (para o n-ésimo valor)
}
```

Código 3.7

```
public static int Fibonacci (int n) // função Fibonacci iterativa, recebe um valor inteiro como parâmetro.
{
    int k, fant=0, fpos=1, f;
    if ((n==0) || (n==1)) return n; // os dois casos base: F(0)=0 e F(1)=1.
    for (k=2; k<=n; k++) {
        f=fant+fpos;
        fant=fpos;
        fpos=f;
    }
    return f; // retorna o valor do n-ésimo número da sequência de Fibonacci
}
```

Código 3.8

```
public static void MoveTorre (int n, char Orig, char Dest, char Aux)
// função principal recursiva da Torre Hanoi
{
    if (n==1)
        MoveDisco(n, Orig, Dest);           // caso base ou condição de parada (tem apenas um disco).
    else {
        MoveTorre(n-1, Orig, Aux, Dest);    // casos recursivos
        MoveDisco(n, Orig, Dest);
        MoveTorre(n-1, Aux, Dest, Orig);
    }
}

public static void MoveDisco (int disco, char Orig, char Dest)
// função que apenas mostra o movimento efetuado.
{
    System.out.println("\nMovimento: Disco " + disco + " de " + Orig + " --> " + Dest);
}
```

Python

Código 3.1

```
def pot2(n):
    k=1
    pot=1
    if n==0:
        return 1
    else:
        while k<=n:
            pot = pot * 2
            k=k+1
        return pot
```

Código 3.2

```
def pot2(n):
    if n==0:
        return 1
    else:
        return 2 * pot2(n-1)
```

Código 3.3

```
def fatorial(n):  
    fat=1  
    if n==0:  
        return fat  
    else:  
        while n>0:  
            fat = fat * n  
            n-=1  
        return fat
```

Código 3.4

```
def fatorial(n):  
    if n==0:  
        return 1  
    else:  
        return n*fatorial(n-1)
```

Código 3.5

```
def fatorial(n):  
L1  if n==0:  
L2      return 1  
    else:  
L3      return n*fatorial(n-1)
```

Código 3.6

```
def Fibonacci(n):  
    if n==0 or n==1:  
        return n  
    else:  
        return Fibonacci(n-1) + Fibonacci(n-2)
```

Código 3.7

```
def Fibonacci(n):
    k=2
    fant=0
    fpos=1
    f=0
    if n==0 or n==1:
        return n
    while k<=n:
        f=fant+fpos
        fant=fpos
        fpos=f
        k+=1
    return f
```

Código 3.8

```
def MoveTorre(n,Orig,Dest,Aux):
    if n==1:
        MoveDisco(n,Orig,Dest)
    else:
        MoveTorre(n-1, Orig, Aux, Dest)
        MoveDisco(n,Orig,Dest)
        MoveTorre(n-1,Aux,Dest,Orig)

def MoveDisco(disco,Orig,Dest):
    print "Movimento: Disco %d de %s --> %s" %(disco,Orig,Dest)
```



Para fixar

1. Faça uma função recursiva para calcular o máximo divisor comum (MDC) de dois números inteiros não negativos, n e m , usando o algoritmo de Euclides:

$$mdc(m,n)=\begin{cases} mdc(n,m) & \text{se } n > m, \\ m & \text{se } n = 0, \\ mdc(n,(m \bmod n)) & \text{se } n > 0. \end{cases}$$

Vamos lá, você consegue!



Papo técnico

Você pode medir o desempenho dos algoritmos utilizando funções existentes nas bibliotecas das principais linguagens de programação. Essas funções pegam a hora do sistema. Faça isso antes de chamar a função e imediatamente após seu retorno. A diferença entre os dois valores é o tempo gasto pelo algoritmo (geralmente em milissegundos).



Para saber mais

Você poderá encontrar mais informações sobre algoritmos recursivos em diversos livros, entre eles *Algoritmos: teoria e prática* (Cormen *et al.* 2012) e *Introdução à ciência da computação* (Mokarzel e Soma, 2008).



Navegar é preciso

Com uma linguagem bem-humorada, o site *Learnyou some Erlang* (<http://learnyousomeerlang.com/recursion>) trata o tema de recursão com uma apropriada profundidade. Além disso, existem vários objetos de aprendizagem para a demonstração das implementações de algoritmos recursivos. Um site que traz várias implementações de simulações de algoritmos recursivos, divididos em simples, intermediários e avançados, é o <http://www.animatedrecursion.com/>

Outra animação bem interessante é o da implementação do problema da Torre de Hanoi. Esse objeto de aprendizagem pode ser acessado em <http://www.cut-the-knot.org/recurrence/hanoi.shtml>

Exercícios

1. Faça uma função recursiva para calcular a potência de um número, considerando que o expoente é um número natural. Utilize o método das multiplicações sucessivas.
2. Faça uma função recursiva que converta um número na base 10 para a base 2.
3. Faça uma função recursiva que retorne o maior valor armazenado em um vetor V , contendo n números inteiros.

Glossário

Teste de mesa: metodologia de teste de um algoritmo sem a utilização de um computador. Para tanto, devemos identificar as variáveis envolvidas, criar uma tabela identificando as variáveis, as instruções, os valores atribuídos à variável em cada uma das linhas de código, valores de entrada e de retorno das funções etc. Também devemos percorrer o algoritmo, passo a passo, identificando todas as variações provocadas pelo mesmo.

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- MOKARZEL, F. C.; SOMA, N. Y. *Introdução à ciência da computação*. Rio de Janeiro: Elsevier, 2008.
- SCHILDT, H. C. *completo e total*. 3. ed. São Paulo: Makron Books, 1996.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.



CO QUE
VEM **DEPOIS**

Agora você já sabe como funcionam os algoritmos recursivos, como implementá-los, e também quando usar e não usar essa técnica. Chegou a hora de nos aprofundar nas técnicas de manipulação de estruturas de dados efetivamente. Para começar, vamos entender como podemos recuperar uma informação, estando ela armazenada em determinada ordem ou não. Preparado? Então, bons estudos!

Métodos de busca

“ Enquanto vamos em busca do incerto, perdemos o seguro. ”

TITUS MACCIUS PLAUTUS

Encontrar algo em um mar de informações torna-se cada vez mais difícil. Assim, conhecer as técnicas apropriadas para cada caso é imprescindível no desenvolvimento de algoritmos de busca.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer os métodos de busca mais utilizados;
- entender a importância do arranjo dos dados (principalmente a ordenação) e sua implicação nos métodos de busca;
- conhecer e praticar a implementação dos principais algoritmos de busca.



Para começar

Todos nós fazemos buscas, consciente e inconscientemente, todos os dias. Fazemos isso quando, por exemplo, procuramos um restaurante onde possamos almoçar, ou uma roupa para usar ao longo do dia, ou determinado livro em uma ou mais estantes de uma biblioteca.

Também fazemos buscas mentais, internamente, em nosso cérebro, ao procurar um rosto conhecido ou o significado de uma palavra que aprendemos anteriormente.



Atualmente, guardar informações não é mais um problema. A capacidade das máquinas cresceu muito, e serviços de *cloud computing*, por exemplo, têm auxiliado ainda mais nessa tarefa. O grande problema está no processo de recuperação dessas informações, principalmente quando precisamos recuperar uma informação de qualidade, algo que possamos realmente utilizar da forma como planejamos ou necessitamos.

Vamos tomar como exemplo uma grande base de dados de produtos de uma loja de autopeças. Suponha que o cliente chegue ao balcão e pergunte se a loja tem determinada peça para vender. Geralmente, o nome ou o código do produto é utilizado como o índice ou chave de busca, dado (campo) que compõe o registro do produto e é utilizado para distingui-lo dos demais; portanto, serve de indicador para a realização da busca na base de dados. A

Figura 4.1 ilustra a organização da base de dados, com a tabela de produtos, seu registro e o campo de cada um.

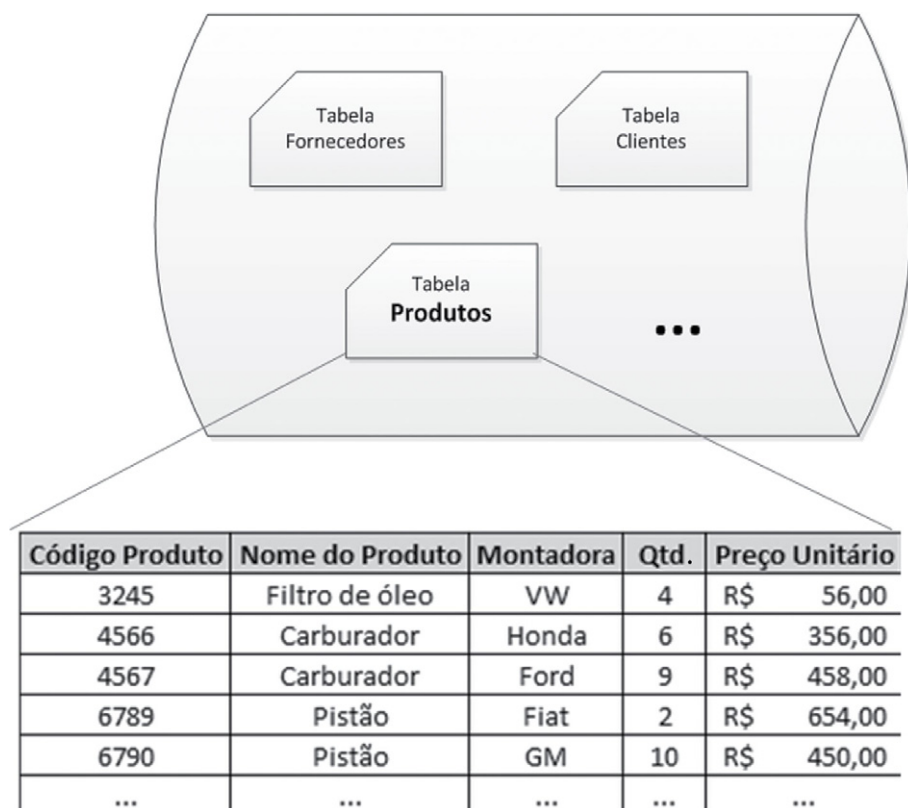


FIGURA 4.1: Estrutura genérica de uma base de dados: tabelas, registros e campos.

Na Figura 4.1, as colunas “Código do produto”, “Nome do produto”, “Montadora”, “Qtd.” e “Preço unitário” são os campos. Cada linha corresponde às informações sobre um produto. A esse conjunto de informações damos o nome de *registro*. Assim, cada linha é um registro. O conjunto de todos os registros dos produtos forma a tabela de produtos. Essa tabela e outras ficam alocadas dentro de uma base de dados.

Atenção

Você verá com detalhes as questões relativas à estruturação das informações em bases de dados nas disciplinas específicas da área de Banco de Dados. Por ora, os conceitos de *tabela*, *registro* e *campo* serão suficientes para as implementações dos algoritmos de busca.



Voltando ao balcão: quando o cliente pergunta sobre determinado produto – por exemplo, “carburador da Ford” –, o funcionário vai até um sistema, busca o nome ou código do produto e retorna ao cliente, dizendo: “Sim, temos! E o preço é R\$ 458,00”.

Aqui surgem algumas dúvidas: como o funcionário encontrou o produto especificado? Quanto tempo se passou até que o funcionário o encontrasse e respondesse ao cliente?

Possivelmente, o funcionário acessou a base de dados através de um sistema automatizado que contava com uma ferramenta de busca. Assim, inseriu a informação passada pelo cliente em um local específico no sistema, e o produto, com suas respectivas informações, foi encontrado e exibido na tela do terminal de consulta.

Isso faz com que perguntemos: como o sistema encontrou essa informação? Possivelmente, utilizando um algoritmo de busca.

E como pode ser implementado um algoritmo de busca?

Bem, pensemos em um conjunto de informações mais simples, como um vetor de 10 elementos (números inteiros), conforme ilustrado na [Figura 4.2](#).

4	78	12	3	65	21	34	77	98	11
---	----	----	---	----	----	----	----	----	----

FIGURA 4.2: Um vetor com 10 números inteiros.

Se alguém lhe perguntasse se no vetor da [Figura 4.2](#) existe o valor 34, você certamente responderia que sim. E poderia ser detalhista e completar dizendo que ele está na sétima posição.

Você conseguiria descrever o método que utilizou para encontrar o valor 34 no vetor?

A resposta mais frequente é: “Corri os olhos pelo vetor, da esquerda para a direita, verificando cada um dos valores, até encontrar o valor procurado: 34.”

Você conseguiria descrever essa operação de busca no formato de um algoritmo (em português estruturado)? Vamos lá, tente!



Dica

Para desenvolver esse algoritmo, pense na sequência de passos que você, até de forma inconsciente, realizou para encontrar determinado número no vetor. Lembre-se de que cada detalhe é muito importante!

E então? Conseguiu?

Possivelmente, seguindo os passos descritos, você deve ter desenvolvido algo semelhante ao representado no algoritmo a seguir.

Algoritmo (em português estruturado)

Supondo um vetor $v[]$, um total de elementos, n , e o valor a ser encontrado, k , teríamos:

- percorra o vetor (da primeira à última posição);
- para cada elemento do vetor, verifique se é igual ao valor a ser encontrado;
- se for igual, retorne com a posição do vetor onde se encontra esse valor;
- caso contrário, continue procurando.

Se você chegou ao final do vetor e não encontrou o valor, retorne o valor -1.

Note que o valor -1 é retornado como uma espécie de sinalizador. Isso é feito porque não existe no vetor uma posição igual a -1. Assim, fica implícito que, ao retornar o -1, o valor não foi encontrado.

Esse método de busca é o mais simples a ser implementado, mas existem muitos outros. Neste capítulo vamos ver apenas os mais simples e utilizados: busca sequencial, busca binária e busca por tabela de espalhamento (*hashing table*). Preparado? Vamos lá!



Conhecendo a teoria para programar

Vamos iniciar com a forma mais simples de buscar um elemento em um arranjo de dados: procurando-o ao longo de todos os seus elementos – no caso de um vetor, verificando todas as suas posições. Esse método é conhecido como *busca sequencial*.

Busca sequencial

Como mencionado, para exemplificar essa forma de busca, vamos utilizar uma estrutura de dados do tipo vetor. Para tanto, o algoritmo de busca sequencial vai percorrer todo o vetor, do início ao fim, comparando o valor que se busca com cada elemento do vetor.

Se durante a busca a comparação for positiva, ou seja, se o valor for encontrado, a posição do vetor será retornada. Caso a comparação chegue ao

final do vetor, isso significa que o valor buscado não foi encontrado. Como uma estratégia nesse caso, o valor retornado geralmente é -1, pois o vetor não tem uma posição com esse valor.

Nós fizemos a descrição desse algoritmo em português estruturado. Vamos agora implementá-lo em linguagem C. Uma possibilidade de implementação pode ser a descrita no Código 4.1. Para tanto, faremos a implementação de uma função que receba como parâmetros um vetor $v[]$ de números inteiros, um número inteiro n , que indica a quantidade de elementos nesse vetor, e por fim outro valor inteiro k , que seria o valor que se busca encontrar no vetor $v[]$. A função retorna a posição do elemento do vetor $v[]$ igual a k (se existir) ou -1, se o valor não for encontrado. Vamos ao código!

Código 4.1

```
int buscaSeq (int v[], int n, int k)
{
    int i;
    for (i=0; i<n; i++) {           // percorre todo o vetor v[]
        if (v[i]==k) return i;      // se encontra um valor igual, retorna a posição i
    }
    return -1;                     // caso o valor não seja encontrado, retorna o valor -1
}
```

Para testar a função *buscaSeq()* apresentada no Código 4.1, poderíamos utilizar o vetor da [Figura 4.2](#). Assim, se passássemos como parâmetro o vetor dessa figura, o número 10 representando o total de elementos do vetor e o número 34 como chave de busca, teríamos como retorno o valor 6, ou seja, a posição do número 34 no vetor.

Atenção

Lembre-se de que nos vetores em algumas linguagens, como C e Java, as posições iniciam em zero. Por esse motivo, o número 34, estando na sétima posição, retorna o valor 6.



Caso o valor que se pretenda procurar for o número 71, o valor de retorno da função *buscaSeq()* será -1, pois esse valor não existe no vetor da [Figura 4.2](#). Nesse caso, ao analisar a função de busca sequencial, veremos que as 10 posições do vetor serão visitadas e comparadas com o valor-chave.

Se, por acaso, os valores do vetor estivessem ordenados (do menor para o maior), existiria uma forma de busca sequencial mais eficiente? Sim, pois poderíamos percorrer o vetor, do menor para o maior, até o ponto em que o valor buscado fosse menor que o valor do *i-ésimo* elemento do vetor. Nesse caso, quando o elemento buscado fosse menor que o valor do *i-ésimo* elemento, a função poderia retornar o valor -1 (não existe valor igual à chave de busca *k*).

Vamos tomar novamente o vetor da [Figura 4.2](#) e organizar seus elementos em ordem crescente. O resultado pode ser visto na [Figura 4.3](#).

Uma possibilidade de otimização do Código 4.1 (busca sequencial), supondo que os elementos do vetor estão organizados em ordem crescente, poderia ser o que é apresentado no Código 4.2.

3	4	11	12	21	34	65	77	78	98
---	---	----	----	----	----	----	----	----	----

FIGURA 4.3: Vetor de 10 elementos inteiros organizados em ordem crescente.

Código 4.2

```
int buscaSeq (int v[], int n, int k)
{
    int i;
    for (i=0; i<n; i++) {        // percorre todo o vetor v[]
        if (v[i]==k) return i;    // se encontra um valor igual, retorna a posição i
        if (v[i]>k) return -1;    // se o i-ésimo element do vetor for maior que a chave k, retorna -1 (não existe igual).
    }
    return -1;                  // caso o valor não seja encontrado, retorna o valor -1
}
```

Observe que, no caso do vetor da [Figura 4.3](#), se fizéssemos a busca tentando encontrar o valor 15 utilizando a função do Código 4.2, o algoritmo

faria no máximo 5 comparações, pois, quando chegasse ao quinto elemento (valor 21), verificaria que esse elemento é maior do que a chave de busca e, assim, retornaria -1 (o valor 15 não existe no vetor).

Utilizando a mesma chave de busca, porém com o algoritmo do Código 4.1, o total de buscas seria igual a 10.



Conceito

Mesmo mostrando maior eficiência para alguns casos, a busca sequencial, com seus elementos estando ou não ordenados, pode, na melhor das hipóteses, realizar apenas uma comparação, e na pior delas, n comparações, onde n é igual ao número total de elementos no vetor.

Se os elementos do vetor não estiverem ordenados, não teremos outro método de busca, a não ser o sequencial. Mas, se os elementos do vetor estiverem ordenados, existem métodos mais eficientes do que a busca sequencial ou linear. Um deles é conhecido como *busca binária*. Vamos conhecê-lo?

Busca binária

Esse método só funciona se os elementos da estrutura de dados (no caso, vetor) estiverem ordenados. Ele utiliza a ideia de “dividir para conquistar”: a divisão acontece sempre comparando o valor que se busca com o elemento localizado no meio do vetor. Ao se realizar essa comparação, há três possibilidades: (1) o valor desse elemento central é igual à chave de busca; (2) o valor é menor do que a chave de busca; ou (3) o valor é maior do que a chave de busca.

No primeiro caso, basta retornar à posição central e encontramos o valor. No segundo caso, se os elementos do vetor estiverem em ordem crescente e o elemento central for menor do que a chave de busca, isso significa que o valor dessa chave, se existir, estará na parte superior do vetor e a parte inferior (da metade ao início) será totalmente ignorada. O mesmo princípio se aplica ao terceiro caso, mas desta vez o elemento que se busca, se existir, estará na parte inferior do vetor e a parte superior (da metade até o final do vetor) será totalmente ignorada.

Depois dessa primeira divisão, o novo vetor passará a conter apenas os elementos da parte selecionada, e uma nova busca será feita com o elemento central. Ocorrerá uma nova divisão e assim sucessivamente, até que se encontre o valor procurado ou se retorne um valor de divisão igual a zero

(ou seja, não existem mais elementos no processo de divisão do vetor). Nesse caso, não existe valor igual à chave de busca.

Uma possibilidade de implementação desse método de busca pode ser o do Código 4.3, a seguir.

Código 4.3

```
int buscaBin (int v[], int n, int k)
{
    int inicio=0;                // início do vetor ou de parte do vetor (primeiro elemento)
    int fim=n-1;                // final do vetor ou de parte do vetor (último elemento)
    int centro;                 // posição central do vetor
    while (inicio <= fim) {      // enquanto existir elementos no vetor...
        centro=(inicio+fim)/2;   // recebe a posição central do vetor
        if (k == v[centro]) return centro; // caso (1) – encontrou o valor
        else if (k > v[centro]) // caso (2) – o valor do elemento central é menor que k
            inicio=centro+1;     // nesse caso (2) o novo vetor passa a ser a parte superior.
        else                    // caso (3) – o valor do elemento central é maior que k
            fim=centro-1;        // nesse caso (3) o novo vetor passa a ser a parte inferior.
    }
    return -1;                  // caso o valor não seja encontrado, retorna o valor -1
}
```

Ao executar essa função passando como parâmetro o vetor da [Figura 4.3](#), com 10 elementos e chave de busca igual a 34 ($k = 34$), teríamos esta sequência de execução:

```
Vetor v[] = {3,4,11,12,21,34,65,77,78,98}
inicio=0/fim=9/centro=4
```

Depois da primeira comparação, identifica-se que o valor 34 pode estar na parte superior do vetor inicial. Assim, um novo *looping* é iniciado no comando *while*:

```
Vetor v[] = {3,4,11,12,21,34,65,77,78,98}
inicio=5/fim=9/centro=7
```


Após a segunda comparação, verifica-se que o valor 34 pode estar na parte inferior desse novo vetor. Assim, um novo *looping* é iniciado:

```
Vetor v[] = {3,4,11,12,21,34,65,77,78,98}
inicio=5/fim=6/centro=5
```

Nessa nova comparação, o valor 34 é encontrado na posição 5 (6º elemento) do vetor e, assim, retornado com sucesso.

Esquemáticamente, teríamos a sequência ilustrada na [Figura 4.4](#).

Você deve ter notado que, a cada *looping*, o vetor é dividido ao meio. Iniciou com 10 elementos, depois foi reduzido para 5, depois para 2. Nesse processo, reduziu-se gradativamente o espaço de busca até encontrar o elemento procurado ou chegar a um vetor vazio (caso em que não existe o valor procurado no vetor), e a velocidade de busca foi acelerada exponencialmente em relação ao método de busca sequencial.

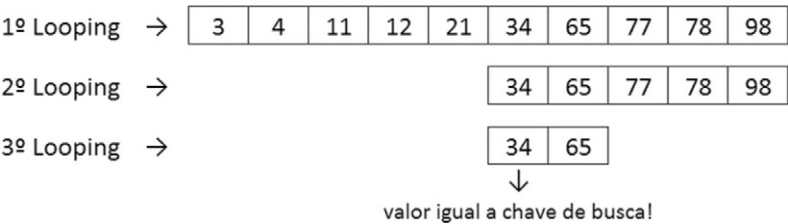


FIGURA 4.4: Sequência de execuções do algoritmo de busca binária.

Além disso, você deve ter notado que, a cada iteração do algoritmo de busca binária, um novo vetor era sugerido para realizar essa busca. Em virtude disso, poderíamos realizar a implementação do algoritmo de busca binária de forma recursiva, passando, a cada execução, um novo vetor (parte) como parâmetro. Embora a implementação não recursiva seja mais eficiente e mais adequada para esse tipo de algoritmo, a implementação recursiva é mais elegante e sucinta. O Código 4.4 ilustra uma possível implementação da função de busca binária recursiva.

Faremos uma pequena modificação na chamada da função: em vez de passar a quantidade de elementos do vetor, passaremos a posição inicial e a final.

Código 4.4

```
int buscaBinRec (int v[], int inicio, int fim, int k)
{
    int centro; // posição central do vetor
    while (inicio <= fim) { // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2; // recebe a posição central do vetor
        if (k == v[centro]) return centro; // caso (1) – encontrou o valor
        else if (k > v[centro]) // caso (2) – o valor do elemento central é menor que k
            return buscaBinRec(v, centro+1, fim, k); // caso (2) o novo vetor passa a ser a parte superior
        else // caso (3) – o valor do elemento central é maior que k
            return buscaBinRec(v, inicio, centro-1, k); // caso (3) o novo vetor é a parte inferior.
    }
    return -1; // caso o valor não seja encontrado, retorna o valor -1
}
```

Agora que você já conhece os dois métodos de busca mais básicos, é hora de partir para uma implementação mais sofisticada: a busca utilizando tabelas de dispersão ou tabelas de espalhamento (em inglês, *hashing tables*).

Tabelas de dispersão

As tabelas de dispersão, ou tabelas *hashing*, consistem no armazenamento de cada elemento em determinado endereço, calculado a partir da aplicação de uma função sobre a chave de busca. Matematicamente, teríamos o seguinte:

$$e = f(c)$$

onde e é o endereço, c é a chave de busca, e $f(c)$ é a função que tem como entrada a chave de busca.

Assim, o processo de pesquisa sobre elementos organizados dessa forma é similar a um acesso direto ao elemento pesquisado. A [Figura 4.5](#) ilustra essa busca utilizando tabelas de dispersão.

A eficiência da pesquisa nesse tipo de organização dos dados depende da função de cálculo do endereço ($f(c)$). A função ideal seria aquela que pudesse gerar um endereço diferente para cada elemento da tabela (chave de busca). Entretanto, isso é praticamente inviável, principalmente pela dinâmica de atualização dos dados e aumento da quantidade de elementos na tabela.

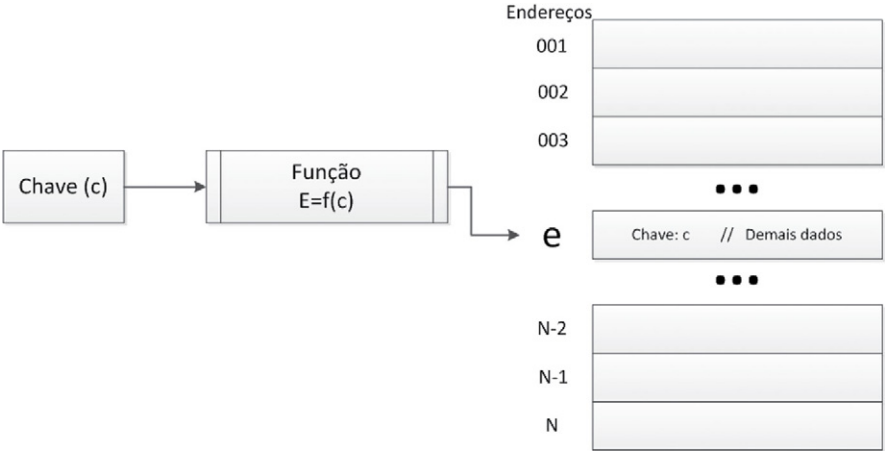


FIGURA 4.5: Processo de busca e organização dos dados em uma tabela de dispersão (*hashing table*).

Vamos entender esse conceito trabalhando com algo concreto. Supondo os dados constantes no vetor da [Figura 4.6](#), acrescido da posição de cada elemento, teríamos:

Endereço →	0	1	2	3	4	5	6	7	8	9
Valores →	30	11	72	83	44	65	86	17	58	9

FIGURA 4.6: Um vetor de 10 elementos inteiros organizados com sua posição.

Pensando nessa tabela e na organização dos dados, poderíamos sugerir uma função de dispersão que tivesse como entrada o valor do elemento (chave de busca), e a função retornaria o endereço no vetor. Na amostra de dados, uma função possível seria:

$$e(c)=(c \bmod 10)$$

onde *mod* corresponde ao resto da divisão inteira da chave *c* por 10 (número de elementos da tabela). Podemos verificar se a função de dispersão proposta é eficiente testando alguns valores ([Tabela 4.1](#)).

Como pode ser visto na tabela acima, a função de dispersão proposta é efetiva para o acesso direto aos valores constantes na tabela da [Figura 4.6](#).

Entretanto, o problema acontece quando existe possibilidade de atualização de valores. Por exemplo, vamos imaginar que queiramos mudar o valor da posição 0 de 30 para 91. A função não é mais eficiente, pois, se aplicarmos utilizando como chave o valor 91, teremos o endereço igual a 1. Como podemos observar, o endereço 1 já está ocupado pelo valor 11.

TABELA 4.1: Aplicação dos valores das chaves de busca sobre a função de dispersão para cálculo do endereço

Chave (c)	$c \bmod 10$	e	Acesso direto?
30	$30 \bmod 10$	0	Ok! $\rightarrow v[0] = 30$
44	$44 \bmod 10$	4	Ok! $\rightarrow v[4] = 44$
9	$9 \bmod 10$	9	Ok! $\rightarrow v[9] = 9$
72	$72 \bmod 10$	2	Ok! $\rightarrow v[2] = 72$

A esse fenômeno de dois valores distintos resultarem em um mesmo endereço, quando aplicados a uma função de dispersão, chamamos de *colisão*. Trata-se de um fenômeno comum que pode ser tratado de diversas formas.

Para entender melhor o processo de colisão no mundo real, considere o seguinte: se você possui um *smartphone*, certamente tem um aplicativo de gerenciamento de contatos. Nesse aplicativo, existem várias maneiras de acessar os contatos; uma delas é escolher a letra inicial do nome. Quando você escolhe determinada letra, são exibidos todos os nomes que começam com ela. Essa é uma forma de entendermos uma tabela de dispersão. A letra inicial do nome é a entrada de uma função, que separa todos os nomes que iniciam com aquela mesma letra. Depois disso, a busca pelo nome desejado fica bem mais rápida. A [Figura 4.7](#) ilustra essa implementação e o controle das colisões.

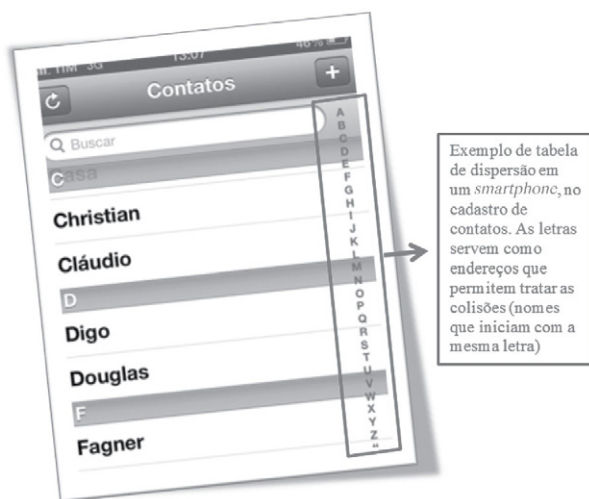


FIGURA 4.7: Tabela de dispersão e o tratamento das colisões em listas de contatos em *smartphones*.

Como já foi dito, existem várias formas de trabalhar as colisões. Uma das mais utilizadas é a implementação de listas encadeadas, estruturas de dados que utilizam alocação dinâmica na memória, a partir do endereço base. Por exemplo, pensando no vetor da [Figura 4.6](#), poderíamos alocar outros números tomando como base uma dispersão de 10 áreas diferentes e utilizando a mesma função para cálculo do endereço. As colisões seriam tratadas com alocação dinâmica dos elementos por listas encadeadas. A [Figura 4.8](#) ilustra essa ideia.

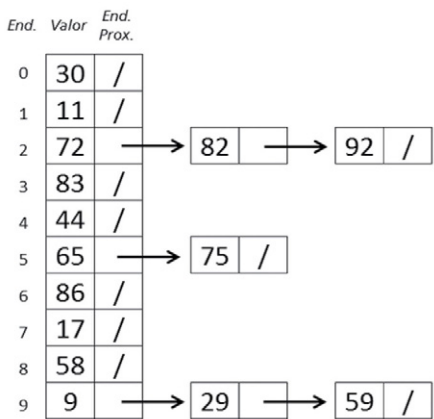


FIGURA 4.8: Tabela de dispersão e o tratamento das colisões com listas encadeadas.

Note que, quando se acessa determinado endereço (dado pela função de dispersão), pode existir um número, nenhum ou uma lista deles. Geralmente, essa lista encadeada de números é mantida em ordem crescente, o que pode facilitar o processo de busca e otimizar o processo.

Atenção



A implementação de estruturas de dados por meio de listas encadeadas será vista no Capítulo 7. Fique atento!

Você deve saber que existem muitos outros métodos e algoritmos de busca. Mais adiante, veremos, algoritmos de busca em listas encadeadas e árvores.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação Java e Python.

Java

Código 4.1

```
public static int buscaSeq (int v[], int n, int k)
{
    int i;
    for (i=0; i<n; i++) {           // percorre todo o vetor v[]
        if (v[i]==k) return i;      // se encontra um valor igual, retoma a posição i
    }
    return -1;                     // caso o valor não seja encontrado, retoma o valor -1
}
```

Código 4.2

```
public static int buscaSeq (int v[], int n, int k)
{
    int i;
    for (i=0; i<n; i++) {           // percorre todo o vetor v[]
        if (v[i]==k) return i;      // se encontra um valor igual, retoma a posição i
        if (v[i]>k) return -1;      // se o i-ésimo elemento do vetor > chave k, retoma -1 (não existe igual).
    }
    return -1;                     // caso o valor não seja encontrado, retoma o valor -1
}
```

Código 4.3

```

public static int buscaBin (int v[], int n, int k)
{
    int inicio=0;                // início do vetor ou de parte do vetor (primeiro elemento)
    int fim=n-1;                 // final do vetor ou de parte do vetor (último elemento)
    int centro;                  // posição central do vetor
    while (inicio <= fim) {      // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2; // recebe a posição central do vetor
        if (k == v[centro]) return centro; // caso (1) – encontrou o valor
        else if (k > v[centro])      // caso (2) – o valor do elemento central é menor que k
            inicio=centro+1;         // nesse caso (2) o novo vetor passa a ser a parte superior.
        else                        // caso (3) – o valor do elemento central é maior que k
            fim=centro-1;            // nesse caso (3) o novo vetor passa a ser a parte inferior.
    }
    return -1;                   // caso o valor não seja encontrado, retorna o valor -1
}

```

Código 4.4

```

public static int buscaBinRec (int v[], int inicio, int fim, int k)
{
    int centro;                  // posição central do vetor
    while (inicio <= fim) {      // enquanto existir elementos no vetor...
        centro=inicio+(fim-inicio)/2; // recebe a posição central do vetor
        if (k == v[centro]) return centro; // caso (1) – encontrou o valor
        else if (k > v[centro])      // caso (2) – o valor do elemento central é menor que k
            return buscaBinRec(v, centro+1, fim, k); // caso (2) o novo vetor passa a ser a parte superior.
        else                        // caso (3) – o valor do elemento central é maior que k
            return buscaBinRec(v, inicio, centro-1, k); // caso (3) o novo vetor é a parte inferior.
    }
    return -1;                   // caso o valor não seja encontrado, retorna o valor -1
}

```

OU pode ser usado a seguinte função:

```

|
System.out.println(Arrays.binarySearch(Array, Valor));

```

Python

Código 4.1

```
def buscaSeq(v,n,k):  
    i=0  
    while i<n:  
        if v[i]==k:  
            return i  
        i+=1  
    return -1
```

Código 4.2

```
def buscaSeq(v,n,k):  
    i=0  
    while i<n:  
        if v[i]==k:  
            return i  
        if v[i]>k:  
            return -1  
        i+=1  
    return -1
```

Código 4.3

```
def buscaBin (v,n,k):  
    inicio=0  
    fim=n-1  
    centro=0  
    while inicio<=fim:  
        centro=inicio+(fim-inicio)/2  
        if k==v[centro]:  
            return centro  
        elif k> v[centro]:  
            inicio=centro+1  
        else:  
            fim=centro-1  
    return -1
```


Código 4.4

```
def buscaBinRec(v, inicio, fim, k):  
    centro=0  
    while inicio<=fim:  
        centro=inicio+(fim-inicio)/2  
        if k==v[centro]:  
            return centro  
        elif k>v[centro]:  
            return buscaBinRec(v, centro+1, fim, k)  
        else:  
            return buscaBinRec(v, inicio, centro-1, k)  
    return -1;
```



Para fixar

1. No código de busca binária abaixo estão faltando algumas informações (“##”). Com base no que você aprendeu, preencha corretamente e faça os comentários pertinentes ao que está acontecendo em cada linha do algoritmo.

```
int buscaBinaria(int v[], int n, int k) {  
    int inicio = ##;  
    int fim = ##;  
    while (inicio ## fim-1) {  
        centro = (inicio+fim)/2;  
        if (v[centro] ## k) return centro;  
        else if (v[centro] ## k) inicio=centro;  
        else fim = centro;  
    }  
    return ##;  
}
```

Vamos lá, você consegue!



Para saber mais

Você poderá encontrar mais informações sobre algoritmos de busca em *Algoritmos: teoria e prática* (Cormen *et al*).



Navegar é preciso

Na internet, existem muitos *sites* e páginas que tratam o tema de recursão em computação.

Um deles é o site da Wikipedia (http://pt.wikipedia.org/wiki/Algoritmo_de_busca), que define a expressão *algoritmo de busca*. Além de interessante, apresenta uma história sobre o assunto e alguns links sobre temas relacionados, como *pesquisa binária*, *pesquisa sequencial*, *pesquisa por interpolação*, *busca com retrocesso*, entre outros. Existem vários objetos de aprendizagem para a demonstração das implementações de algoritmos de busca. Selecionamos dois deles para demonstrar a busca sequencial e a busca binária:

- busca sequencial

http://www.cse.ust.hk/learning_objects/array/linearsearch/index.html

- busca binária

http://www.cse.ust.hk/learning_objects/array/binarysearch/index.html

Exercícios

1. Na sua opinião, que problemas poderiam surgir da utilização do código de função de busca a seguir?

```
int Busca (int vet[], int n, int k){  
    int i=0;  
    while (v[i] < k && i < n) ++i;  
    return i;  
}
```

2. Mostre que a função `buscaBin`, a seguir, funciona corretamente.

```
int buscaBin (int vet[], int n, int k){  
    int inicio=0, fim=n, centro;  
    while (inicio < fim)  
        centro=(inicio+fim)/2;  
    if (v[centro] < k) inicio=centro+1;  
    else fim=centro;  
}  
return fim;  
}
```

3. Escreva uma função recursiva de busca binária em que sejam passados apenas dois valores como parâmetros: o vetor ($v[]$) e a chave de busca (k).

Glossário

Cloud Computing: computação na nuvem/em nuvem. Trata-se de um modelo de computação (processamento, armazenamento e aplicações) utilizado em qualquer lugar e em qualquer equipamento, por meio da internet.

Referências bibliográficas

- CORMEN, T. H. C. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- MOKARZEL, F. C.; SOMA, N. Y. *Introdução à ciência da computação*. Rio de Janeiro: Elsevier, 2008.
- SCHILDT, H. C. *completo e total*. 3. ed. São Paulo: Makron Books, 1996.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.
- VELOSO, P. et al. *Estrutura de dados*. 4. ed. Rio de Janeiro: Campus, 1986.



QUE VEM DEPOIS...

Agora que você já conhece os principais algoritmos de busca e entende a importância da ordenação de dados, chegou a hora de se aprofundar na forma como esses dados podem ser ordenados. No Capítulo 5 vamos ver os principais algoritmos de ordenação e suas principais aplicações. Preparado? Então, bons estudos!

Métodos de ordenação

“ Organização é o princípio de tudo. Mantê-la significa competência. ”

ALFREDO VALENTE JUNIOR

Dados e informações são matérias-primas essenciais para a geração do conhecimento. Se, no passado, as empresas pecavam pela falta de dados e informações, principalmente devido à ausência de meios de armazenamento, atualmente, com esses meios ultrapassando os *yottabytes* de capacidade, elas pecam pelo excesso. Portanto, organizar dados e informações é fundamental para rápidas tomadas de decisões dentro das empresas.

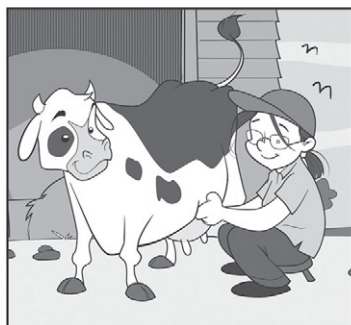
OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender a importância da ordenação e da organização dos dados em ambientes computacionais;
- conhecer alguns dos principais métodos de ordenação interna de dados;
- criar algoritmos para ordenação de dados, previamente armazenados em vetores.



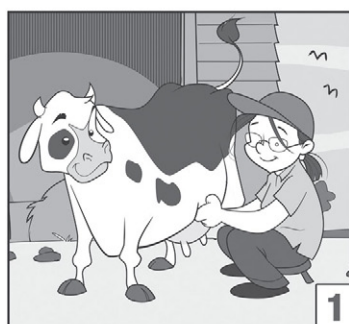
Para começar



Em nosso cotidiano, nos deparamos com situações que, para serem entendidas ou resolvidas, requerem o uso da lógica.

Conceitualmente, a *lógica* trata da correção (ou ordenação) do pensamento. Portanto, quando os fatos ou fatores que motivaram determinada situação apresentam-se de forma ordenada, seu entendimento, ou sua solução, pode ser mais lógico e rápido. Uma pessoa organizada tem como prática colocar as coisas sempre em certa ordem. Quando construímos um algoritmo, procuramos sempre “ordenar” as instruções segundo determinada lógica.

Observe a figura a seguir, da esquerda para a direita, de cima para baixo. Nessa primeira leitura, tente entender qual informação (ou conhecimento) essa figura está passando.



Conseguiu?

Se a resposta foi SIM, você certamente fez mais de uma leitura, procurou seguir a numeração das figuras, ou seja, usou um processo de ordenação (pelos números) para entender o significado da figura.

Considerando cada figura um dado, você ordenou todos os dados e obteve uma informação. Se, contudo, considerou cada figura uma informação, você ordenou as informações e conseguiu obter um conhecimento.

Agora, repita esse processo lendo a figura a seguir.



Agora ficou mais fácil, não é mesmo?

Bastou uma leitura para você conhecer ou se informar sobre o *ciclo de processamento do leite* até chegar ao consumidor final.

Em ambientes computacionais, a ordenação de dados e informações, efetuada por meio de algoritmos/programas que, na maioria das vezes, são de alta complexidade, é algo bastante comum e de extrema importância, principalmente dentro dos contextos atuais dos sistemas de informação baseados em computadores.

Atualmente, muitos aplicativos, tanto específicos (Sistemas Integrados de Gestão - ERP) quanto genéricos (MS-Excel®, Windows Explorer® etc.),

disponibilizam interfaces para a classificação de dados e informações. Essas funcionalidades à disposição dos usuários, que com um simples processo de seleção e alguns cliques de *mouse* conseguem visualizar dados e informações em várias ordens, são na realidade algoritmos/programas já desenvolvidos e testados que usam *métodos de ordenação*.

A [Figura 5.1](#), por exemplo, mostra, na barra de menu da interface do aplicativo Microsoft® Windows Explorer®, a opção “Exibir”, na qual temos a funcionalidade da ordenação (classificação) dos arquivos e diretórios do computador. Nessa interface, podemos ver o conteúdo de um diretório de programas e fontes em linguagem Pascal (...\\ProgFPC\\Fontes), classificado por data de modificação, em ordem ascendente.

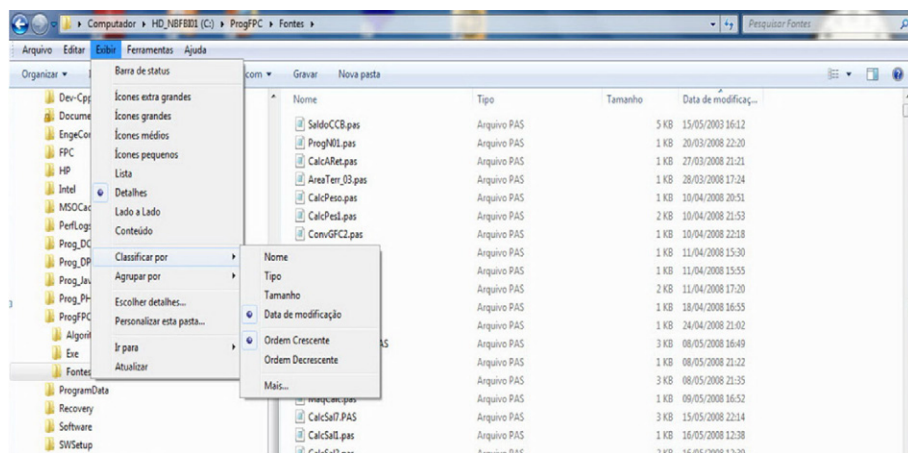
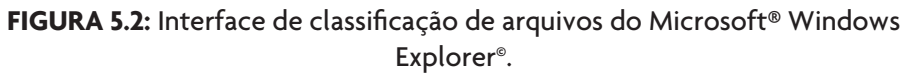


FIGURA 5.1: Interface de classificação de arquivos do aplicativo Microsoft® Windows Explorer®.

Se, contudo, você deseja visualizar os arquivos desse diretório por nome em ordem crescente, basta selecionar essas opções na barra de menu da interface, conforme mostrado na [Figura 5.2](#). Pronto! Os arquivos já aparecem ordenados, como você solicitou.

Selecione um diretório do seu computador e classifique seu conteúdo por tipo ou tamanho do arquivo, usando a funcionalidade de ordenação de arquivos do Microsoft® Windows Explorer®. Foi muito simples porque você encontrou pronto e disponível o programa de ordenação ou classificação.

Assim, neste capítulo, em vez de usar programas prontos de ordenação de dados, você vai aprender a construir algoritmos/programas desse tipo. Para tanto, conhecerá alguns métodos de ordenação ou classificação de dados, e as situações a que esses métodos se aplicam.



Atenção



Embora as palavras *ordenação* e *organização* sejam frequentemente utilizadas como sinônimos, em computação, quando falamos em ordem de dados e informações, pensamos em *classificação*, ao passo que ao falarmos em sistemas de informação, pensamos em *organização*. Portanto, antes de passarmos para os conceitos sobre métodos de ordenação, vamos entender esses dois termos.

Atenção

Em processamento de dados, a ordenação ou classificação é o processo no qual se determina a ordem de apresentação das entradas de uma estrutura de dados, de modo que eles obedeçam à sequência ditada por um ou mais componentes ou campos. Os componentes determinantes da ordenação são denominados *campos-chaves de classificação*.

Portanto, dependendo da estrutura de dados que estamos querendo ordenar (homogênea ou heterogênea), para ordenar dados e/ou informações devemos determinar um ou mais *campo-chaves de classificação*, e em seguida sua *ordem de classificação* (ascendente ou descendente).

Nós podemos ordenar uma planilha MS-Excel® ou uma tabela de dados considerando uma ou mais colunas contidas nesses arquivos. Quando, segundo critérios predefinidos, ordenamos os dados de uma coluna, podemos localizar rapidamente os valores nela contidos e utilizá-lo para diversas finalidades, principalmente para tomadas de decisão.

É possível, por exemplo, ordenar uma planilha de preços de um produto primeiro pelo grupo a que ele pertence e, em seguida, pelo seu código; nesse caso, certamente queremos conhecer o preço desse produto tendo em mãos seu código e o grupo a que ele pertence. Podemos também ordenar essa planilha pelo conteúdo da coluna “preço do produto” em ordem descendente (do maior para o menor) e, em seguida, pela coluna “código do produto”; nesse caso, provavelmente estamos procurando pelos produtos mais caros.

Organizar significa colocar cada coisa no seu devido lugar, arrumar. Se as coisas (dados) estão ordenadas, a organização consiste em estruturá-las da melhor forma possível, para que possam atender às reais necessidades de sua recuperação e utilização. Infere-se, a partir desse conceito, que o processo de ordenação antecede o da organização.



Dica

Ao construir algoritmos/programas para a ordenação de dados e de informações, além do método de ordenação, é necessário determinar previamente o campo-chave ou campos-chave de classificação, bem como sua ordem.

Esses conceitos iniciais sobre ordenação/organização ficaram claros? Então, você já deve estar pensando em ordenar e organizar alguma coisa, certo?

Se a resposta foi SIM, provavelmente você está olhando ao seu redor e tentando descobrir o que pode ser ordenado e organizado, e, principalmente, como isso pode ser feito. Porém, como nosso objetivo é a ordenação de dados, sugiro que você comece a pensar em algumas estruturas de dados e em como ordená-las.

É isso o que vamos começar a estudar a partir deste momento.

Preparado? Vamos em frente!



Conhecendo a teoria para programar

Para entender a importância da ordenação de dados, imagine que você está consultando a lista telefônica da cidade de São Paulo e tentando encontrar o número do telefone de uma pessoa conhecida. Imagine também que a lista não traz os nomes dos assinantes em ordem alfabética. Difícil, não é mesmo?

Você deve ter aprendido em disciplinas como Fundamentos de Tecnologia da Informação ou Introdução à Computação, que o processamento de dados comercial aconteceu entre o final da década de 50 e o início da década de 60. Nessa época já havia uma grande preocupação dos analistas de sistemas e programadores de computador com a ordenação de dados, pois, a partir do momento em que o computador passou a ser utilizado para fins comerciais, percebeu-se que os dados por ele processados e disponibilizados aos usuários deveriam estar em formato que atendessem às expectativas desses indivíduos.

Uma dessas expectativas era quanto à *entrada, ao processamento e à saída de dados de forma ordenada*, pois isso permitiria aos usuários organizar melhor seus processos de trabalho e, em última análise, contribuiria para a organização da empresa.

Os primeiros computadores usados para fins comerciais no início da década de 1960 tinham o cartão perfurado como único meio para o armazenamento de dados (observe a [Figura 5.3](#)). Cada coluna desse cartão armazenava um caractere (numérico, alfabético ou alfanumérico), portanto, ele podia ser utilizado para estruturar dados em até 80 colunas, que era sua capacidade máxima de armazenamento.

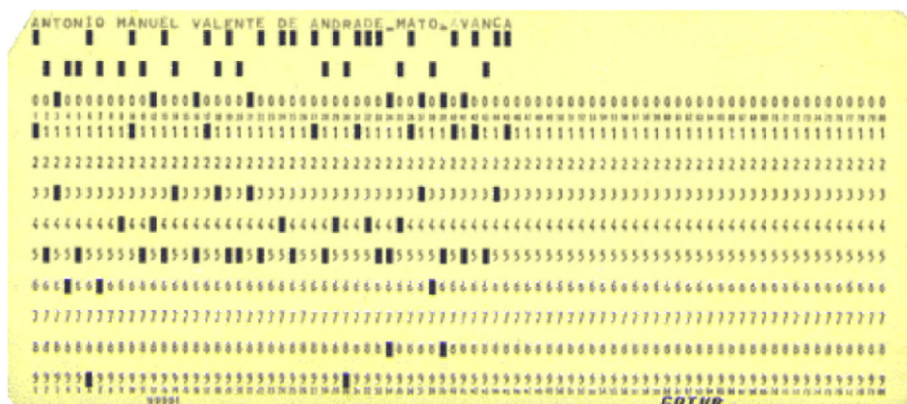


FIGURA 5.3: Cartão perfurado de 80 colunas.

A [Figura 5.4](#) mostra um cartão com uma estrutura de dados heterogênea (registro) de um cliente. Nele é possível identificar um campo “Código”, ocupando as colunas de 1 a 5; um campo “Nome do cliente”, da coluna 6 a 22; um campo “Cidade e sigla do Estado”, da coluna 49 a 60, e assim por diante.

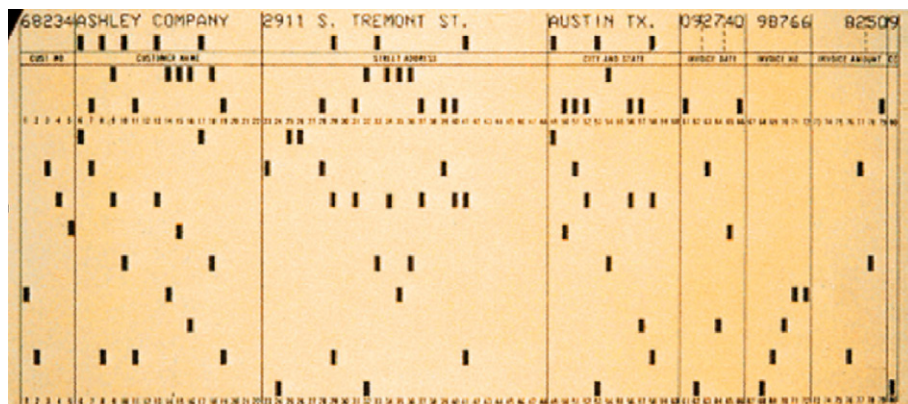


FIGURA 5.4: Cartão perfurado de 80 colunas contendo dados de um cliente.

Você deve estar imaginando: se o cartão perfurado era o único meio de armazenamento de dados utilizado pelos computadores nessa época, então como ocorria a ordenação dos dados nele registrados?

Lembre-se de que, hoje, ordenar dados é um processo relativamente simples em termos de execução de programa de classificação. Existem discos magnéticos que funcionam como memória auxiliar e são utilizados por esses programas como “áreas de trabalho de classificação”, uma vez que ordenam dados executando várias vezes as estruturas lógicas de repetição, principalmente aquelas com variável de controle numérico das repetições (*For... Do*). Vale ressaltar que, no início da década de 1960, os meios magnéticos de armazenamento, ou seja, os discos rígidos, ainda não existiam. Nessa época, a ordenação de dados contidos em cartões perfurados era feito por um dispositivo classificador desses cartões. No caso da IBM®, o dispositivo era o *Card Sorter IBM 0083*®, mostrado na [Figura 5.5](#).

Atenção

Somente na metade da década de 1960 a IBM® apresentou o primeiro disco magnético para armazenamento dados - o IBM RAMAC 305® -, que tinha a capacidade de armazenar 5 MB em 50 discos de 2 pés de diâmetro.





FIGURA 5.5: Classificador de cartão perfurado de 80 colunas - IBM® 0083.

O processo de classificação de cartões perfurados era lento, pois era mecânico. Todavia, o conceito de classificação utilizado nessas máquinas continua válido até hoje, nos algoritmos de classificação, ou seja, a classificação é feita *byte a byte* (antes, era feito coluna a coluna), sendo usadas para isso estruturas lógicas de repetição, além de memória principal e auxiliar para a classificação.



Papo técnico

Desde a década de 1960, os algoritmos/programas de ordenação de dados ficaram conhecidos como *SORT* (do inglês *sort*, que, em português, significa “ordenar, classificar”).

Ambiente de Classificação

Um processo de classificação pode ser menos ou mais eficiente, dependendo das características do meio onde estiverem armazenados os dados a serem classificados, ou seja, se estiverem em memória primária ou auxiliar.

Se os dados estiverem em memória primária, então o tempo de acesso a qualquer um de seus endereços será constante e igual para todos, em qualquer momento. Se, porém, os dados estiverem armazenados em memória auxiliar ou secundária (por exemplo, em disco rígido), precisarão ser transferidos para a memória principal para, em seguida, ser processados pela CPU. Esse processo de transferência normalmente é feito por blocos de dados (vários endereços), ou seja, vários endereços são transferidos por vez.



Dica

Para que o processo de ordenação de dados contidos em memória auxiliar seja eficiente, os dados nessa memória devem estar organizados de tal forma que o bloco transferido a cada acesso ao disco seja efetivamente processado pela CPU.

Os algoritmos de ordenação de dados devem levar em consideração o ambiente de classificação, pois cada um deles possui uma característica que poderá influenciar sua complexidade e eficiência.



Papo técnico

O processo de classificação de dados contidos inteiramente na memória principal é denominado *classificação interna*.

O processo de classificação de dados que não cabe inteiramente na memória principal e que, portanto, os dados estão contidos em memória secundária é denominado *classificação externa*.

Neste capítulo, vamos propor e desenvolver algoritmos/programas de ordenação considerando o processo de classificação interna.

Métodos de classificação interna

Considerando os objetivos propostos no início deste capítulo, serão considerados os seguintes métodos de ordenação:

- por troca (*BubbleSort*);
- por troca (*QuickSort*);
- por seleção direta (*SelectionSort*).

Método de ordenação por troca - Método da bolha
ou *Bubblesort*

É um método de classificação caracterizado por efetuar ordenação comparando sucessivamente pares de elementos e mudando-os de posição quando se apresentam fora da ordem desejada.

O *BubbleSort* é um dos métodos de classificação mais simples. É conhecido como “método da bolha” porque usa a estratégia de “borbulhar” o maior elemento (o elemento de maior valor) para o final do arranjo (ou estrutura) de dados que está sendo ordenado.

Como exemplo, vamos considerar o seguinte arranjo, que contém 7 elementos:

9	8	6	12	4	3	7
---	---	---	----	---	---	---

Esse método consiste em efetuar uma varredura no arranjo, da esquerda para a direita, *comparando pares de elementos consecutivos e trocando* de lugar os que estão fora de ordem (levando sempre o elemento de maior valor para o final da lista).

Varredura 1

E[1]	E[2]	E[3]	E[4]	E[5]	E[6]	E[7]	Troca
9	8	6	12	4	3	7	1 e 2
8	9	6	12	4	3	7	2 e 3
8	6	9	12	4	3	7	Não
8	6	9	12	4	3	7	4 e 5
8	6	9	4	12	3	7	5 e 6
8	6	9	4	3	12		6 e 7
8	6	9	4	3	7	12	Fim da 1ª varredura

Ao final da primeira varredura, o maior elemento (12) encontra-se alocado em sua posição definitiva no arranjo. Deixamos esse elemento de lado e iniciamos a segunda varredura no “subarranjo” com os elementos E[1],..., E[m-1], onde m = 7.

Varredura 2

							Troca
8	6	9	4	3	7	12	1 e 2
6	8	9	4	3	7	12	não
6	8	9	4	3	7	12	3 e 4
6	8	4	9	3	7	12	4 e 5
6	8	4	3	9	7	12	5 e 6
6	8	4	3	7	9	12	Fim da 2ª Varredura

Vamos repetindo as varreduras em subarranjos cada vez menor ($[m-2]$, $[m-3]$, ...); a última varredura é feita no subarranjo $E[1]$, $E[2]$.

A seguir, temos no Código 5.1, escrito em linguagem C, a *estrutura lógica do método* de ordenação da bolha, ou *BubbleSort*.

Código 5.1

```
// Método de ORDENACAO da Bolha ou BubbleSort
for (IndVet=0; IndVet < 6; IndVet++)
    for (IndVet1=0; IndVet1 < 6; IndVet1++)
        if (Vet_Ch[IndVet1] > Vet_Ch[IndVet1+1]) // Compara o elemento anterior com o posterior, do arranjo de dados.
        {
            Aux = Vet_Ch[IndVet1]; // Se anterior é maior, troca de posição com o posterior, dentro do arranjo.
            Vet_Ch[IndVet1] = Vet_Ch[IndVet1+1];
            Vet_Ch[IndVet1+1] = Aux; // Se anterior é maior, troca de posição com o posterior, dentro do arranjo.
        }
```




Papo técnico

Complexidade do método BubbleSort

Por ser um método de ordenação muito simples, o *BubbleSort* não apresenta bom desempenho de execução quando a lista a ser ordenada contém muitos itens de dados. A complexidade desse método é de ordem quadrática $\theta(n^2)$, pois são necessários n^2 passos de processamento para cada número n de elementos que serão ordenados. Portanto, recomenda-se esse método somente para fins de aprendizagem acadêmica, não indicando-o para aplicações profissionais.

Método de ordenação por troca - *QuickSort*

Tanto o *BubbleSort* quanto o *QuickSort* são algoritmos que utilizam o método de ordenação por troca dos elementos que estão sendo ordenados. Todavia, a principal diferença entre eles é que *enquanto o BubbleSort troca pares de elementos consecutivos, o QuickSort compara e troca pares de elementos distantes*, característica esta que acelera o processo de ordenação.



Papo técnico

O algoritmo do *QuickSort* foi originalmente apresentado por G. A. R. Hoare, em 1960. É o método de ordenação interna mais rápido que se conhece para uma ampla variedade de situações, e, provavelmente, o mais utilizado por programadores profissionais.

Esse método se baseia em um conceito importante na construção de algoritmos, que é “dividir para conquistar”.

O método *QuickSort* consiste em escolher determinado elemento da estrutura que será ordenada – a que vamos denominar *estrutura original* – e alocá-lo em sua posição correta dentro da estrutura que resultará da

ordenação – a que vamos denominar *estrutura ordenada*. Dessa forma, ocorre uma partição (divisão da estrutura original em três subestruturas), ou seja:

1. uma subestrutura 1, que conterà todos os elementos menores ou iguais ao elemento alocado na estrutura ordenada;
2. uma subestrutura 2, que conterà somente o elemento alocado inicialmente;
3. uma subestrutura 3, que conterà todos os elementos maiores ou iguais ao elemento alocado.

Após o particionamento (*Dividir*), as subestruturas 1 e 3 são ordenadas por chamadas recursivas ao método de ordenação *QuickSort*.

As subestruturas 1, 2 e 3 são ordenadas localmente, portanto, não há necessidade de promover sua junção. A estrutura de dados está ordenada (*Conquistar*) ao término das chamadas recursivas.

Atenção



O processo de particionamento descrito acima é repetido para as subestruturas 1 e 3, e assim sucessivamente. Quando uma subestrutura apresentar um número de elementos menor ou igual a um número preestabelecido, a ela é aplicado um método simples de classificação em vez de partição.

Para exemplificar esse método de ordenação, consideremos um vetor - **Vet_Ch** - com 8 elementos, contendo dados numéricos inteiros sem sinal (chaves de classificação). Esse vetor tem a seguinte estrutura: Vet_Ch[Iniseg..FimSeg].

8	12	9	7	6	11	10	4	Vet_Ch
1	2	3	4	5	6	7	8	Vet_End

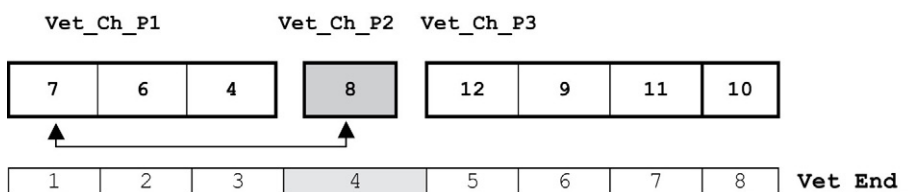
O primeiro passo é *dividir* esse vetor e escolher o elemento mais à esquerda (que contém valor igual a 8) como *elemento pivô* a partir do qual o vetor será particionado.



Atenção

Assim como escolhemos o elemento mais à esquerda como elemento alocado, ou elemento pivô da divisão do vetor, poderíamos ter escolhido o elemento mais à direita, ou outro qualquer dentro da estrutura.

Em seguida, escolhemos o elemento de Vet_Ch, que vai trocar de posição com o elemento pivô (no caso, o resultado da divisão dos elementos do arranjo foi = 4, então o *quarto elemento de Vet_Ch vai trocar de posição*). Após a execução dessa divisão, teremos o vetor Vet_Ch particionado em três subestruturas:



A subestrutura Vet_Ch_P1 contém todos os elementos de Vet_CH *menores ou iguais a 8 (valor do elemento alocado)*. Temos, então, Vet_Ch[IniSeg..Val_EA - 1].

A subestrutura Vet_Ch_P2 contém somente o valor do elemento alocado de Vet_Ch. Temos, então, Vet_Ch[4].

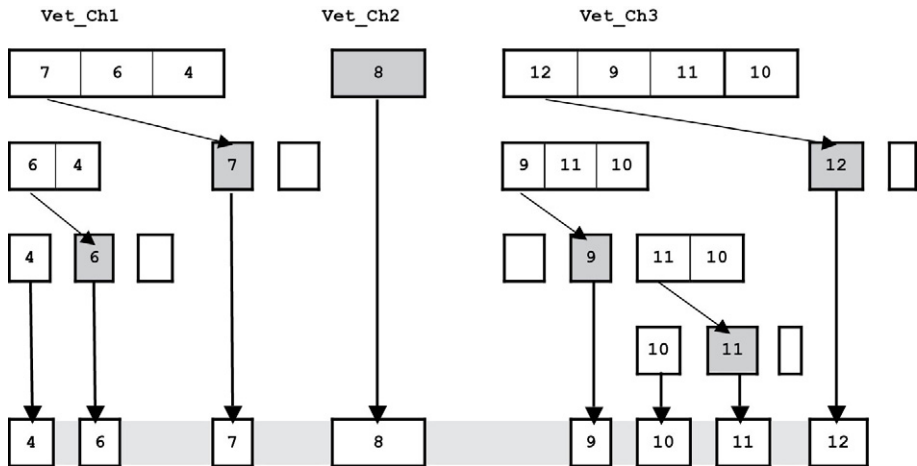
A subestrutura Vet_Ch_P3 contém todos os elementos de Vet_CH *maiores ou iguais a Val_EA*. Temos, então, Vet_Ch[Val_EA + 1, FimSeg].

Observação

IniSeg aponta para o primeiro elemento de Vet_Ch, e FimSeg aponta para o último elemento. No exemplo, temos IniSeg=1 e FimSeg=8.

Após a partição (*dividir*) da estrutura Vet_Ch, *aplica-se recursivamente o método de ordenação QuickSort* até que as subestruturas resultantes tornem-se unitárias (contenham somente um elemento) ou vazias.

Ao final da aplicação do método de ordenação denominado *QuickSort* (método da troca e partição), temos a estrutura de dados (Vetor) ordenado, conforme mostrado a seguir.



Mostraremos a seguir, em linguagem C, os procedimentos que permitem a ordenação da estrutura de dados do nosso exemplo - Vet_Ch pelo método do *QuickSort*. Estamos associando a essa estrutura um vetor com o endereço dos seus elementos - Vet_End.

8	12	9	7	6	11	10	4	Vet_Ch
1	2	3	4	5	6	7	8	Vet_End

Primeiro, mostraremos no Código 5.2, escrito em linguagem C, o procedimento: *particao_ED*, em linguagem C, que contém a *estrutura lógica do particionamento do arranjo de dados*. Em seguida, mostraremos no Código 5.3, também em linguagem C, o procedimento que contém o método de ordenação por troca - *QuickSort* -, em que temos a recursividade desse procedimento.

Código 5.2

```
// Método QuickSort - Particionamento de estrutura/arranjo de dados

void particao_ED (int IniSeg, int FimSeg)
{
    int IniSeg1, FimSeg1, // Início e Fim do segmento
    Ch, // Contem o valor do elemento Pivô
    Ender; // Contem o Índice do elemento Pivô
    bool Esq; // Verdadeiro quando troca elemento a esquerda do vetor
    IniSeg1 = IniSeg;
    FimSeg1 = FimSeg;
    Ch = Vet_Ch[IniSeg]; // Primeiro elemento do arranjo

    Ender = Vet_End[IniSeg];
    Esq = true;

    while (IniSeg1 < FimSeg1)
    {
        if (Esq)
            if (Ch > Vet_Ch[FimSeg1])
            {
                Vet_Ch[IniSeg1] = Vet_Ch[FimSeg1];
                Vet_End[IniSeg1] = Vet_End[FimSeg1];
                IniSeg1 = IniSeg1 + 1;
                Esq = false;
            }
            else
                FimSeg1 = FimSeg1 - 1;
        else
            if (Ch < Vet_Ch[IniSeg1])
            {
                Vet_Ch[FimSeg1] = Vet_Ch[IniSeg1];
                Vet_End[FimSeg1] = Vet_End[IniSeg1];
                FimSeg1 = FimSeg1 - 1;
                Esq = true;
            }
            else
                IniSeg1 = IniSeg1 + 1;
    } // Fim do While

    Vet_Ch[IniSeg1] = Ch;
    Vet_End[IniSeg1] = Ender;
    ElemPart = IniSeg1;
} // Fim do procedimento particao_ED
```

Código 5.3

```
// Método QuickSort

void QuickSort(int IniSeg, int FimSeg)
{
    int NumElem;

    NumElem = (FimSeg - IniSeg) / 2; //Determina o elemento particionador da estrutura de dados (índice do vetor)

    if (FimSeg > IniSeg)
        if ((FimSeg - IniSeg) > NumElem)
        {
            particao_ED(IniSeg, FimSeg);
            QuickSort(IniSeg, ElemPart-1); // Recursividade de QuickSort
            QuickSort(ElemPart+1, FimSeg); // Recursividade de QuickSort
        }
    // Fim da Procedure: QuickSort
}
```



Papo técnico

Complexidade do método QuickSort

É um método de ordenação rápido, por causa da sua vantagem significativa em termos de eficiência: funciona muito bem com uma lista grande de itens de dados e ordena a lista no próprio local, não havendo necessidade de espaço adicional de armazenamento.

A pequena desvantagem apresentada por esse método é que seu pior desempenho é similar à média de desempenho dos outros aqui apresentados - $\theta(n^2)$. Todavia, é importante ressaltar que casos de pior desempenho são raros.

O desempenho no caso médio e no melhor caso é $\theta(n \log n)$, observado na ordenação de listas randômicas, em que a partição é mais eficiente.

Geralmente, o método *QuickSort* apresenta organização mais eficiente, sendo altamente recomendado para ordenar listas de qualquer tamanho.

Método de ordenação por seleção direta - *Selectionsort*

É um método de ordenação no qual, a cada passo, é feita uma varredura da estrutura de dados que contém os elementos ainda não selecionados, *determinando o elemento dessa estrutura, que tem o menor valor*. Esse elemento é colocado, então, na primeira posição da estrutura de dados, por troca. Assim, a estrutura que está sendo ordenada fica com um elemento a menos.

Esse processo é repetido até que a estrutura de dados fique com um só elemento, que é o de maior valor. Nesse momento, temos a estrutura ordenada.

Para exemplificar, consideremos uma estrutura de dados (vetor) - Vet_Ch – com 8 elementos, contendo números inteiros sem sinal (cada elemento contém uma chave de classificação).

Vamos considerar também um Vetor - Vet_End – contendo o endereço (posição) dos elementos no vetor - Vet_Ch. Ao se ordenar Vet_Ch, a relação dos elementos desse vetor com os elementos de Vet_End será mantida.

Na varredura o terceiro elemento de Vet_Cn, contendo valor = 6, é o de menor valor e será colocado na primeira posição de Vet_Ch, por troca.

15	21	6	14	31	13	11	9	Vet_Ch
1	2	3	4	5	6	7	8	Vet_End

6	21	15	14	31	13	11	9	Vet_Ch
3	2	1	4	5	6	7	8	Vet_End

6	9	15	14	31	13	11	21	Vet_Ch
3	8	1	4	5	6	7	2	Vet_End

6	9	11	14	31	13	15	21	Vet_Ch
3	8	7	4	5	6	1	2	Vet_End

Essa varredura continua até que o vetor fique com apenas um elemento, que é o de maior valor, no caso o elemento de endereço = 5, contendo o valor = 31.

Pronto! O vetor está ordenado.

6	9	11	13	14	15	21	31	Vet_Ch
3	8	7	6	4	1	2	5	Vet_End

A seguir, temos no Código 5.4, escrito em linguagem C, a estrutura lógica que permite a ordenação de uma estrutura de dados, pelo método da seleção direta - *SelectionSort*.

Código 5.4

```
// Método de ORDENAÇÃO por Seleção Direta - SelectionSort

for (IndVet=0; IndVet < 7; IndVet++)
{
    Menor = IndVet;
    for (IndVet1 = IndVet+1; IndVet1 < 8; IndVet1++)
        if (Vet_Ch[IndVet1] < Vet_Ch[Menor])
            Menor = IndVet1;

    Ch = Vet_Ch[IndVet];
    Vet_Ch[IndVet] = Vet_Ch[Menor];
    Vet_Ch[Menor] = Ch;
    Ender = Vet_End[IndVet];
    Vet_End[IndVet] = Vet_End[Menor];
    Vet_End[Menor] = Ender;
}
```



Papo técnico

Complexidade do método SelectionSort

Conforme já estudado, esse método consiste em vasculhar repetidamente uma lista de itens de dados, selecionando um elemento de cada vez e colocando-o na posição correta da sequência. Portanto, funciona muito bem com listas pequenas. Todavia, sua complexidade é semelhante à do método *BubbleSort* - $\theta(n^2)$ – que exige n^2 números de passos para cada n elementos da lista.

Portanto, o *SelectionSort* é adequado apenas para listas de dados em que poucos itens estejam em ordem aleatória.

Para aplicarmos os métodos de ordenação apresentados neste capítulo, vamos usar como exemplo a situação a seguir. Diariamente, o funcionário do almoxarifado de uma empresa necessita de um relatório (ou tela) que mostre o saldo atual de cada produto em estoque. A lista de que ele dispõe atualmente está ordenada por código do produto, conforme mostrado na tela abaixo.

```
<< Estoque de Produtos >> - Ordenado pelo Codigo do Produto
```

Cod. Produto	Saldo em Estoque
1001	50
1002	30
1003	-10
1004	85
1005	100
1006	-25
1007	-40
1008	45

```
>>>> Pressione qualquer tecla para continuar. . .
```

Para melhorar seu trabalho diário, ele necessita de uma lista ordenada pelo saldo em estoque, em ordem crescente.

Implementação em c do método de ordenação por troca - BubbleSort

Para atender à necessidade desse funcionário, desenvolveremos um programa que implementa o método *BubbleSort* para classificar os arranjos de dados de códigos de produtos e saldo em estoque. Para mostrar o conteúdo dos arranjos de dados antes e depois da ordenação, implementamos um procedimento (modularização) denominado `mostra_ED`.



Dica

Consulte o Capítulo 12 do livro “*Algoritmos e programação de computadores*”, também de nossa autoria, para relembrar os conceitos sobre modularização de algoritmos (procedimento/função).

O Código 5.5 a seguir, escrito em linguagem C, está aplicando o método do Código 5.1, ordenando a estrutura de dados – `Vet_Estoque` – e, ao mesmo tempo, mantendo a relação com a estrutura de dados `Vet_Produto`, por meio da ordenação dos seus elementos.

Código 5.5

```
// Aplicação do método BubbleSort

#include <stdio.h>           // Biblioteca com operações de I/O (Input/Output)
#include <string.h>          // Biblioteca com operações sobre String (cadeia de caracteres)
#include <stdlib.h>

int Vet_Estoque[8] = {50, 30, -10, 85, 100, -25, -40, 45};           // Arranjo de dados - Estoque de Produto
int Vet_Produto[8] = {1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008}; // Arranjo de Códigos de Produto
int IndVet, IndVet1, Aux, Aux1;
char palavra[8];

// mostra_ED - procedimento para mostrar o conteúdo de Vet_Produto, Vet_Estoque, ANTES e APÓS sua ordenação

void mostra_ED(char palavra[])
{
    system("cls");
    printf("\n");
    printf("\n %s", "<< Estoque de Produtos >> - ", palavra);
    printf("\n");
    printf("\n %s \n", "Cod. Produto    Saldo em Estoque\n");

    for (IndVet=0; IndVet < 8; IndVet++)
        printf("      %i %i \n", Vet_Produto[IndVet], "      ", Vet_Estoque[IndVet]);

    printf("\n");
    printf("\n >>>> ");
    system("pause");
}

int main()           // função principal
{
    strcpy(palavra, "Ordenado pelo Codigo do Produto ");
    mostra_ED(palavra);           // Chamada de Procedimento

    // Método de ORDENACAO da Bolha ou Bubblesort

    for (IndVet=0; IndVet < 7; IndVet++)
        for (IndVet1=0; IndVet1 < 7; IndVet1++)

    // Compara elemento anterior com o posterior. Se maior permuta os elementos dentro da Estrutura ou arranjo de dados

    if (Vet_Estoque[IndVet1] > Vet_Estoque[IndVet1+1])
    {
        Aux = Vet_Estoque[IndVet1];
        Aux1 = Vet_Produto[IndVet1];
        Vet_Estoque[IndVet1] = Vet_Estoque[IndVet1+1];
        Vet_Produto[IndVet1] = Vet_Produto[IndVet1+1];
        Vet_Estoque[IndVet1+1] = Aux;
        Vet_Produto[IndVet1+1] = Aux1;
    }

    strcpy(palavra, "Ordenado pelo Saldo em Estoque ");
    mostra_ED(palavra);           // Chamada de Procedimento
    return 0;
}
```

A execução desse código mostra os arranjos de dados antes da ordenação, ou seja, o estoque de produtos apresenta-se ordenado pela coluna “Código do produto”. Após a ordenação, usando o método *BubbleSort*, será mostrada a tela abaixo, que é a lista de estoque de produtos ordenada pela coluna “Saldo em estoque”.

```
<< Estoque de Produtos >> - Ordenado pelo Saldo em Estoque
Cod. Produto    Saldo em Estoque
1007            -40
1006            -25
1003            -10
1002            30
1008            45
1001            50
1004            85
1005            100
>>>> Pressione qualquer tecla para continuar. . .
```

Implementação em c do método de ordenação por troca - QuickSort

Considerando o exemplo anterior, desenvolvemos o Código 5.6, a seguir, também em linguagem C, implementando o método de ordenação contido nos Códigos 5.2 e 5.3 - *QuickSort*.

Código 5.6

```
// Aplicação do método Quicksort
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int Vet_Estoque[8] = {50,30,-10,85,100,-25,-40,45}; // Arranjo de dados: Saldo em Estoque
int Vet_Produto[8] = {1001,1002,1003,1004,1005,1006,1007,1008}; // Arranjo de dados: Códigos Produtos
int IndVet,
    IniSeg, // Início do arranjo de dados
    FimSeg, // Fim do arranjo de dados
    ElemPart, // Elemento particionador do arranjo
    Aux;
char palavra[8];

void mostra_ED(char palavra[]) // mostrar o conteúdo de Vet_Estoque e Vet_Produto, ANTES e APÓS sua ordenação
{
    system("cls");
    printf("\n");
    printf("\n %s", "<< Estoque de Produtos >> - ", palavra);
    printf("\n");
    printf("\n %s \n", "Cod. Produto Saldo em Estoque\n");

    for (IndVet=0; IndVet < 8; IndVet++)
        printf(" %i %i \n", Vet_Produto[IndVet], " ", Vet_Estoque[IndVet]);

    printf("\n");
    printf("\n>>>> ");
    system("pause");
}

void particao_ED (int IniSeg, int FimSeg) // Particionamento do arranjo de dados pelo método Quicksort
{
    int IniSeg1,
        FimSeg1, // Fim do segmento - variável local
        Ch, // Contem o valor do elemento Pivô
        Ender; // Contem o índice do elemento Pivô
    bool Esq; // Verdadeiro quando troca elemento a esquerda do vetor
    IniSeg1 = IniSeg;
    FimSeg1 = FimSeg;
    Ch = Vet_Estoque[IniSeg]; // Elemento Alocado - Primeiro elemento do arranjo
    Ender = Vet_Produto[IniSeg];
    Esq = true;
}
```

```

while (IniSeg1 < FimSeg1)
{
    if (Esq)
        if (Ch > Vet_Estoque[FimSeg1])
        {
            Vet_Estoque[IniSeg1] = Vet_Estoque[FimSeg1];
            Vet_Produto[IniSeg1] = Vet_Produto[FimSeg1];
            IniSeg1 = IniSeg1 + 1;
            Esq = false;
        }
        else
            FimSeg1 = FimSeg1 - 1;
    else
        if (Ch < Vet_Estoque[IniSeg1])
        {
            Vet_Estoque[FimSeg1] = Vet_Estoque[IniSeg1];
            Vet_Produto[FimSeg1] = Vet_Produto[IniSeg1];
            FimSeg1 = FimSeg1 - 1;
            Esq = true;
        }
        else
            IniSeg1 = IniSeg1 + 1;
} //Fim do While

Vet_Estoque[IniSeg1] = Ch;
Vet_Produto[IniSeg1] = Ender;
ElemPart = IniSeg1;

} //Fim do procedimento particao_ED

void quicksort(int IniSeg, int FimSeg)
{
    int NumElem;

    NumElem = (FimSeg - IniSeg) / 2; // Determina Elemento Particionador do arranjo

    if (FimSeg > IniSeg)
        if ((FimSeg - IniSeg) > NumElem)
        {
            particao_ED(IniSeg, FimSeg);
            quicksort(IniSeg, ElemPart-1); // Recursividade de Quicksort
            quicksort(ElemPart+1, FimSeg); // Recursividade de Quicksort
        }
    } //Fim da Procedura: Quicksort

int main() //função principal
{
    strcpy(palavra, "Ordenado peloCodigo do Produto ");
    mostra_ED(palavra); // Chamada de Procedimento

// Método de ORDENACAO Quicksort

    quicksort(0, 7);

    strcpy(palavra, "Ordenado pelo Saldo em Estoque ");
    mostra_ED(palavra); // Chamada de Procedimento

    return 0;
}

```

Após a execução do Código 5.6, assim como na execução do Código 5.5, a lista de estoque será mostrada primeiro em ordem de código do produto; após a ordenação, usando o método *QuickSort*, será mostrada a tela a seguir, que é a lista de estoque de produtos ordenada pela coluna “Saldo em estoque”.

```
<< Estoque de Produtos >> - Ordenado pelo Saldo em Estoque
Cod. Produto    Saldo em Estoque
1007            -40
1006            -25
1003            -10
1002            30
1008            45
1001            50
1004            85
1005            100
>>>> Pressione qualquer tecla para continuar. . .
```

Implementação em c do método de ordenação por seleção direta - SelectionSort

Por fim, o mesmo exemplo da classificação da lista de estoque será implementado pelo Código 5.7, a seguir, aplicando o método de ordenação contido no Código 5.4 - *SelectionSort*.

A exemplo dos códigos anteriores, este ordena a estrutura de dados – Vet_Estoque – e, ao mesmo tempo, mantém a relação com a estrutura de dados – Vet_Produto – por meio da ordenação de seus elementos.

A tela abaixo é resultante da execução do Código 5.7, e apresenta a lista de estoque ordenada pela coluna: Saldo em Estoque.

Código 5.7

```
// Aplicação do método SelectionSort

#include <stdio.h>           // Biblioteca com operações de I/O (Input/Output)
#include <string.h>          // Biblioteca com operações sobre String (cadeia de caracteres)
#include <stdlib.h>

int Vet_Estoque[8]={50,30,-10,85,100,-25,-40,45}; // Arranjo de dados - Saldo em Estoque
int Vet_Produto[8]={1001,1002,1003,1004,1005,1006,1007,1008}; // Arranjo de dados - Códigos de Produtos
int IndVet, IndVet1, Menor, Ch, Ender;
char palavra[8];

void mostra_ED(char palavra[]) // mostrar o conteúdo de Vet_Estoque e Vet_Produto, ANTES e APOS sua ordenação
{
    system("cls");
    printf("\n");
    printf("\n %s", "<< Estoque de Produtos >> - ", palavra);
    printf("\n");
    printf("\n %s \n", "Cod. Produto    Saldo em Estoque\n");
    for (IndVet=0; IndVet < 8; IndVet++)
        printf("      %i %i \n", Vet_Produto[IndVet], "      ", Vet_Estoque[IndVet]);
    printf("\n");
    printf("\n>>>> ");
    system("pause");
}

int main()           // função principal
{
    strcpy(palavra, "Ordenado pelo Codigo do Produto ");
    mostra_ED(palavra);           // Chamada de Procedimento

// Método de ORDENACAO por Seleção Direta - SelectionSort

    for (IndVet=0; IndVet < 7; IndVet++)
    {
        Menor = IndVet;
        for (IndVet1 = IndVet+1; IndVet1 < 8; IndVet1++)

            if (Vet_Estoque[IndVet1] < Vet_Estoque[Menor])
                Menor = IndVet1;           // Troca Elemento do Vetor

        Ch = Vet_Estoque[IndVet];
        Vet_Estoque[IndVet] = Vet_Estoque[Menor];
        Vet_Estoque[Menor] = Ch;
        Ender = Vet_Produto[IndVet];
        Vet_Produto[IndVet] = Vet_Produto[Menor];
        Vet_Produto[Menor] = Ender;

        strcpy(palavra, "Ordenado pelo Saldo em Estoque ");
        mostra_ED(palavra);           // Chamada de Procedimento

    }

    return 0;
}
```



```
<< Estoque de Produtos >> - Ordenado pelo Saldo em Estoque  
Cod. Produto    Saldo em Estoque  
1007            -40  
1006            -25  
1003            -10  
1002            30  
1008            45  
1001            50  
1004            85  
1005            100  
  
>>>> Pressione qualquer tecla para continuar. . .
```



Vamos programar

Nesta seção, vamos inicialmente mostrar a implementação dos Códigos 5.1, 5.2, 5.3 e 5.4 nas linguagens Java e Python, lembrando que esses códigos mostram os métodos de ordenação.

Em seguida, mostraremos programas nas linguagens Java e Python, que implementam os métodos de ordenação do código citados para a solução do nosso exemplo - lista de estoque. Os códigos são:

- 5.5 - implementa o método de ordenação do Código 5.1;
- 5.6 - implementa os métodos de ordenação dos Códigos 5.2 e 5.3;
- 5.7 - implementa o método de ordenação do Código 5.4.

Java

Código 5.1

```
// Método de ORDENACAO da Bolha ou BubbleSort

public class JavaApplication
{
    public static void main(String[] args)
    {
        for (IndVet=0; IndVet < 6; IndVet++)
            for (IndVet1=0; IndVet1 < 6; IndVet1++)
                if (Vet_Ch[IndVet1] > Vet_Ch[IndVet1+1])
                {
                    Aux = Vet_Ch[IndVet1];
                    Vet_Ch[IndVet1] = Vet_Ch[IndVet1+1];
                    Vet_Ch[IndVet1+1] = Aux;
                }
    }
}
```

Código 5.2

```
// Método QuickSort - Particionamento de estrutura/arranjo de dados

public static int particao(int []Vet_Ch, int[] Vet_End, int ini, int fim)
{
    int pivo, topo, i;
    int pivoElem;
    pivo = Vet_Ch[ini];
    pivoElem = Vet_End[ini];
    topo = ini;

    for (i = ini + 1; i <= fim; i++)
    {
        if (Vet_Ch[i] < pivo)
        {
            Vet_Ch[topo] = Vet_Ch[i];
            Vet_Ch[i] = Vet_Ch[topo + 1];
            Vet_End[topo] = Vet_End[i];
            Vet_End[i] = Vet_End[topo + 1];
            topo++;
        }
    }
    Vet_Ch[topo] = pivo;
    Vet_End[topo] = pivoElem;
    return topo;
}
```

Código 5.3

```
// Método QuickSort

public static void quick_sort(int[] Vet_Ch, int[] Vet_End, int ini, int fim)
{
    int meio;
    if (ini < fim)
    {
        meio = particao(Vet_Ch, Vet_End, ini, fim);
        quick_sort(Vet_Ch, Vet_End, ini, meio);
        quick_sort(Vet_Ch, Vet_End, meio + 1, fim);
    }
}
```

Código 5.4

```
// Método de ORDENACAO por Seleção Direta - SelectionSort

public static void SelectSort(int[] Vet_Ch, int[] Vet_End)
{
    int indexMin, aux, indexMinEnd, auxEnd;

    for (int i = 0; i < Vet_Ch.length-1; i++)
    {
        indexMin = indexMinEnd = i;
        for (int j = i + 1; j < Vet_Ch.length; j++)
        {
            if (Vet_Ch[j] < Vet_Ch[indexMin])
            {
                indexMin = indexMinEnd = j;
            }
        }
        if (indexMin != i)
        {
            aux = Vet_Ch[indexMin];
            Vet_Ch[indexMin] = Vet_Ch[i];
            Vet_Ch[i] = aux;
            auxEnd = Vet_End[indexMinEnd];
            Vet_End[indexMinEnd] = Vet_End[i];
            Vet_End[i] = auxEnd;
        }
    }
}
```

Código 5.5

```
// Aplicação do método BubbleSort

public class JavaApplication2 {

    // private static int opcao;

    public static void main(String[] args) {

        int[] Vet_Estoque = {50, 30, -10, 85, 100, -25, -40, 45}; // Arranjo de dados
        int[] Vet_Produto = {1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008}; // Arranjo de dados
        int IndVet, IndVet1, Aux, Aux1;
        String palavra = new String();

        palavra = "Ordenado pelo Codigo do Produto";
        mostra_ED(palavra, Vet_Estoque, Vet_Produto);

        // Método de ORDENACAO da Bolha ou Bubblesort

        for (IndVet = 0; IndVet < 6; IndVet++) {
            for (IndVet1 = 0; IndVet1 < 6; IndVet1++) {
                if (Vet_Estoque[IndVet1] > Vet_Estoque[IndVet1 + 1]) {
                    Aux = Vet_Estoque[IndVet1];
                    Aux1 = Vet_Produto[IndVet1];
                    Vet_Estoque[IndVet1] = Vet_Estoque[IndVet1 + 1];
                    Vet_Produto[IndVet1] = Vet_Produto[IndVet1 + 1];
                    Vet_Estoque[IndVet1 + 1] = Aux;
                    Vet_Produto[IndVet1 + 1] = Aux1;
                }
            }
        }
        palavra = "Ordenado pelo Saldo em Estoque";
        mostra_ED(palavra, Vet_Estoque, Vet_Produto);

        // TODO code application logic here
    }

    public static void mostra_ED(String palavra, int[] Vet_Estoque, int[] Vet_Produto) {
        palavra = "<< Estoque de Produtos >> - " + palavra;
        System.out.println(palavra);
        System.out.println("Cod. Produto Saldo em Estoque");
        for (int i = 0; i < Vet_Estoque.length; i++) {
            System.out.println(Vet_Estoque[i] + " " + Vet_Produto[i]);
        }
    }
}
```

Código 5.6

```
//Aplicação do método Quicksort

public class JavaApplication2 {

    public static void main(String[] args) {

        int[] Vet_Estoque = {50,30,-10,85,100,-25,-40,45}; // Elementos chaves
        int[] Vet_Produto = {1001,1002,1003,1004,1005,1006,1007,1008}; // Endereços do elementos chaves
        String palavra = new String();

        palavra = "Ordenado pelo Codigo do Produto";
        mostra_ED(palavra,Vet_Produto,Vet_Estoque);

        quick_sort(Vet_Estoque,Vet_Produto,0,7);

        palavra = "Ordenado pelo Saldo em Estoque";
        mostra_ED(palavra,Vet_Produto,Vet_Estoque);

    }

    public static void mostra_ED(String palavra, int[] Vet_Estoque, int[] Vet_Produto) {
        palavra = "<< Estoque de Produtos >> - " + palavra;
        System.out.println(palavra);
        System.out.println("Cod. Produto      Saldo em Estoque");
        for (int i = 0; i < Vet_Estoque.length; i++) {
            System.out.println(Vet_Estoque[i] + "          " + Vet_Produto[i]);
        }
    }

    public static void quick_sort(int[] Vet_Estoque, int[] Vet_Produto, int ini, int fim) {
        int meio;

        if (ini < fim) {
            meio = particao(Vet_Estoque, Vet_Produto, ini, fim);
            quick_sort(Vet_Estoque, Vet_Produto, ini, meio);
            quick_sort(Vet_Estoque, Vet_Produto, meio + 1, fim);
        }
    }

    public static int particao(int []Vet_Estoque, int[] Vet_Produto, int ini, int fim) {
        int pivo, topo, i;
        int pivoElem;
        pivo = Vet_Estoque[ini];
        pivoElem = Vet_Produto[ini];
        topo = ini;

        for (i = ini + 1; i <= fim; i++) {

            if (Vet_Estoque[i] < pivo) {
                Vet_Estoque[topo] = Vet_Estoque[i];
                Vet_Estoque[i] = Vet_Estoque[topo + 1];
                Vet_Produto[topo] = Vet_Produto[i];
                Vet_Produto[i] = Vet_Produto[topo + 1];
                topo++;
            }
        }
        Vet_Estoque[topo] = pivo;
        Vet_Produto[topo] = pivoElem;
        return topo;
    }
}
```

Código 5.7

```
// Aplicação do método SelectionSort

public class JavaApplication2 {

    public static void main(String[] args) {

        int[] Vet_Estoque = {50,30,-10,85,100,-25,-40,45};           //Elementos chaves
        int[] Vet_Produto = {1001,1002,1003,1004,1005,1006,1007,1008}; //Endereços do elementos chaves

        String palavra = new String();

        palavra = "Ordenado pelo Codigo do Produto";
        mostra_ED(palavra,Vet_Produto,Vet_Estoque);

        SelectSort(Vet_Estoque,Vet_Produto);

        palavra = "Ordenado pelo Saldo em Estoque";
        mostra_ED(palavra,Vet_Produto,Vet_Estoque);

    }

    public static void mostra_ED(String palavra, int[] Vet_Estoque, int[] Vet_Produto) {
        palavra = "<< Estoque de Produtos >> - " + palavra;
        System.out.println(palavra);
        System.out.println("Cod. Produto      Saldo em Estoque");
        for (int i = 0; i < Vet_Estoque.length; i++) {
            System.out.println(Vet_Estoque[i] + "          " + Vet_Produto[i]);
        }
    }

    public static void SelectSort(int[] Vet_Estoque, int[] Vet_Produto) {
        int indexMin, aux, indexMinEnd, auxEnd;

        for (int i = 0; i < Vet_Estoque.length-1; i++) {
            indexMin = indexMinEnd = i;
            for (int j = i + 1; j < Vet_Estoque.length; j++) {
                if (Vet_Estoque[j] < Vet_Estoque[indexMin]) {
                    indexMin = indexMinEnd = j;
                }
            }
            if (indexMin != i) {
                aux = Vet_Estoque[indexMin];
                Vet_Estoque[indexMin] = Vet_Estoque[i];
                Vet_Estoque[i] = aux;
                auxEnd = Vet_Produto[indexMinEnd];
                Vet_Produto[indexMinEnd] = Vet_Produto[i];
                Vet_Produto[i] = auxEnd;
            }
        }
    }
}
```

Python

Código 5.1

```
// Método de ORDENACAO da Bolha ou BubbleSort

def BubbleSort():
    IndVet=0
    while IndVet<6:
        IndVet1=0
        while IndVet1<6:
            if Vet_Ch[IndVet1] > Vet_Ch[IndVet1+1]: # Compara o elemento anterior com o posterior, do arranjo de dados.
                Aux = Vet_Ch[IndVet1] # Se anterior é maior, troca de posição com o posterior, dentro do arranjo.
                Vet_Ch[IndVet1] = Vet_Ch[IndVet1+1]
                Vet_Ch[IndVet1+1] = Aux # Se anterior é maior, troca de posição com o posterior, dentro do arranjo.
            IndVet1+=1
        IndVet+=1
```

Código 5.2

```
// Método QuickSort - Particionamento de estrutura/arranjo de dados

def particao_ED(IniSeg, FimSeg):
    IniSeg1= IniSeg
    FimSeg1 = FimSeg #Fim do segmento
    Ch = Vet_Ch[IniSeg] # Contém o valor do elemento Pivô, nesse caso o primeiro elemento do arranjo

    Ender = Vet_End[IniSeg] #Contem o índice do elemento Pivô
    Esq = True #Verdadeiro quando troca elemento a esquerda do vetor

    while IniSeg1 < FimSeg1:
        if Esq:
            if Ch > Vet_Ch[FimSeg1]:
                Vet_Ch[IniSeg1] = Vet_Ch[FimSeg1]
                Vet_End[IniSeg1] = Vet_End[FimSeg1]
                IniSeg1 = IniSeg1 + 1
                Esq = False
            else:
                FimSeg1 = FimSeg1 - 1
        if Ch < Vet_Ch[IniSeg1]:
            Vet_Ch[FimSeg1] = Vet_Ch[IniSeg1]
            Vet_End[FimSeg1] = Vet_End[IniSeg1]
            FimSeg1 = FimSeg1 - 1
            Esq = True
        else:
            IniSeg1 = IniSeg1 + 1

    #Fim do While

    Vet_Ch[IniSeg1] = Ch
    Vet_End[IniSeg1] = Ender
    ElemPart = IniSeg1

    #Fim do procedimento particao_ED
```

Código 5.3

```
// Método QuickSort

def quicksort(IniSeg,FimSeg):
    NumElem=(FimSeg - IniSeg) / 2

    if FimSeg > IniSeg:
        if (FimSeg - IniSeg) > NumElem:
            particao_ED(IniSeg,FimSeg)
            quicksort(IniSeg,ElemPart-1)
            quicksort(ElemPart+1,FimSeg)
```

Código 5.4

```
// Método de ORDENACAO por Seleção Direta - SelectionSort

def selectionSort():
    IndVet=0
    while IndVet<7:
        Menor = IndVet
        IndVet1=0
        while IndVet1<8:
            if Vet_Ch[IndVet1] < Vet_Ch[Menor]:
                Menor = IndVet1;
            Ch = Vet_Ch[IndVet];
            Vet_Ch[IndVet] = Vet_Ch[Menor];
            Vet_Ch[Menor] = Ch;
            Ender = Vet_End[IndVet];
            Vet_End[IndVet] = Vet_End[Menor];
            Vet_End[Menor] = Ender;
            IndVet1+=1
        IndVet+=1
```

Código 5.5

```
// Aplicação do método BubbleSort

Vet_Estoque = [50,30,-10,85,100,-25,-40,45] # Arranjo de dados - Estoque de Produto
Vet_Produto = [1001,1002,1003,1004,1005,1006,1007,1008] # Arranjo de Códigos de Produto
IndVet=0
IndVet1=0
Aux=0
Aux1=0
palavra=""

# mostra_ED - procedimento para mostrar o conteúdo de Vet_Produto, Vet_Estoque, ANTES e APOS sua ordenação

def mostra_ED(selpalavra):
    print("%s%s" %("<< Estoque de Produtos >> - ",palavra))
    print("%s" %("Cod. Produto Saldo em Estoque\n"))
    IndVet=0
    while IndVet < 8:
        print(" %i%i" % (Vet_Produto[IndVet], " ", Vet_Estoque[IndVet]))
        IndVet+=1
```



```

    print(">>>> ")

# função principal

palavra="Ordenado pelo Codigo do Produto ";
mostra_ED(palavra);          #Chamada de Procedimento

# Método de ORDENACAO da Bolha ou Bubblesort

IndVet=0
while IndVet < 7:
    IndVet1=0
    while IndVet1 < 7:

        if Vet_Estoque[IndVet1] > Vet_Estoque[IndVet1+1]:
            Aux = Vet_Estoque[IndVet1]
            Aux1 = Vet_Produto[IndVet1]
            Vet_Estoque[IndVet1] = Vet_Estoque[IndVet1+1]
            Vet_Produto[IndVet1] = Vet_Produto[IndVet1+1]
            Vet_Estoque[IndVet1+1] = Aux
            Vet_Produto[IndVet1+1] = Aux1
            IndVet1+=1
        IndVet+=1
    palavra="Ordenado pelo Saldo em Estoque "
    mostra_ED(palavra);          # Chamada de Procedimento

```


Código 5.6

```
// Aplicação do método Quicksort

Vet_Estoque = [50, 30, -10, 85, 100, -25, -40, 45] # Arranjo de dados: Saldo em Estoque
Vet_Produto = [1001, 1002, 1003, 1004, 1005, 1006, 1007, 1008] # Arranjo de dados: Códigos Produtos
IndVet=0
IniSeg=0 # Início do arranjo de dados
FimSeg=0 # Fim do arranjo de dados
ElemPart=0 # Elemento particionador do arranjo
Aux=0
palavra=""

def mostra_ED(palavra): # mostrar conteúdo de Vet_Estoque e Vet_Produto, ANTES e APOS sua ordenação
    print("\n %s%s", "<< Estoque de Produtos >> - ", palavra);
    print("\n");
    print("\n %s \n", "Cod. Produto Saldo em Estoque\n")
    IndVet=0
    while IndVet < 8:
        print(" %i %i \n" % (Vet_Produto[IndVet], Vet_Estoque[IndVet]))
        IndVet+=1

    print(">>> ");

def particao_ED (IniSeg, FimSeg): # Particionamento do arranjo de dados pelo método Quicksort
    IniSeg1=0
    FimSeg1=0 # Fim do segmento - variável local
    Ch=0 # Contém o valor do elemento Pivô
    Ender=0 # Contém o índice do elemento Pivô
    Esq=False # Verdadeiro quando troca elemento a esquerda do vetor
    IniSeg1 = IniSeg
    FimSeg1 = FimSeg

    Ch = Vet_Estoque[IniSeg] # Elemento Alocado - Primeiro elemento do arranjo
    Ender = Vet_Produto[IniSeg]
    Esq = True
    while IniSeg1 < FimSeg1:
        if Esq:
            if Ch > Vet_Estoque[FimSeg1]:
                Vet_Estoque[IniSeg1] = Vet_Estoque[FimSeg1]
                Vet_Produto[IniSeg1] = Vet_Produto[FimSeg1]
                IniSeg1 = IniSeg1 + 1
                Esq = False
            else:
                FimSeg1 = FimSeg1 - 1
        else:
            if Ch < Vet_Estoque[IniSeg1]:
                Vet_Estoque[FimSeg1] = Vet_Estoque[IniSeg1]
                Vet_Produto[FimSeg1] = Vet_Produto[IniSeg1]
                FimSeg1 = FimSeg1 - 1
                Esq = True
            else:
                IniSeg1 = IniSeg1 + 1
```

```

        IniSeg1 = IniSeg1 + 1
        # Fim do While

    Vet_Estoque[IniSeg1] = Ch
    Vet_Produto[IniSeg1] = Ender
    ElemPart = IniSeg1

# Fim do procedimento particao_ED

def quicksort(IniSeg,FimSeg):
    NumElem = (FimSeg - IniSeg) / 2    # Determina Elemento Particionador do arranjo
    if FimSeg > IniSeg:
        if (FimSeg - IniSeg) > NumElem:
            particao_ED(IniSeg,FimSeg)
            quicksort(IniSeg,ElemPart-1)    # Recursividade de Quicksort
            quicksort(ElemPart+1,FimSeg)    # Recursividade de Quicksort
# Fim do procedimento: Quicksort

# função principal
palavra="Ordenado peloCodigo do Produto "
mostra_ED(palavra);    # Chamada de Procedimento
# Metodo de ORDENACAO Quicksort
quicksort(0,7)
palavra="Ordenado pelo Saldo em Estoque "
mostra_ED(palavra);    # Chamada de Procedimento

```

Código 5.7

```
// Aplicação do método SelectionSort

Vet_Estoque=[50,30,-10,85,100,-25,-40,45]           # Arranjo de dados - Saldo em Estoque
Vet_Produto=[1001,1002,1003,1004,1005,1006,1007,1008] # Arranjo de dados - Códigos de Produtos
IndVet=0
IndVet1=0
Menor=0
Ch=0
Ender=0
palavra=""

def mostra_ED(palavra): # mostrar o conteúdo de Vet_Estoque e Vet_Produto, ANTES e APOS sua ordenação
    print("%s%s" %("<< Estoque de Produtos >> - ",palavra))
    print("\n %s \n" %("Cod. Produto   Saldo em Estoque\n"))
    IndVet=0
    while IndVet < 8:
        print("      %i %i \n" %(Vet_Produto[IndVet], "      ", Vet_Estoque[IndVet]))
        IndVet+=1
    print(">>>> ");

# função principal

palavra="Ordenado pelo Código do Produto "
mostra_ED(palavra); # Chamada de Procedimento

# Método de ORDENACAO por Seleção Direta - SelectionSort

IndVet=0
while IndVet < 7:
    Menor = IndVet
    IndVet1 = IndVet+1
    while IndVet1 < 8:
        if Vet_Estoque[IndVet1] < Vet_Estoque[Menor]:
            Menor = IndVet1 # Troca Elemento do Vetor
        Ch = Vet_Estoque[IndVet]
        Vet_Estoque[IndVet] = Vet_Estoque[Menor]
        Vet_Estoque[Menor] = Ch
        Ender = Vet_Produto[IndVet]
        Vet_Produto[IndVet] = Vet_Produto[Menor]
        Vet_Produto[Menor] = Ender
        IndVet1+=1
    IndVet+=1
palavra="Ordenado pelo Saldo em Estoque "
mostra_ED(palavra); # Chamada de Procedimento
```



Para fixar

Considerando os programas desenvolvidos na seção anterior, tente alterá-los para ordenar as estruturas de dados em ordem decrescente.



Dica

Consulte o Capítulo 8 (a partir da página 206) do livro *Algoritmos e programação de computadores* para relembrar os conceitos sobre estruturas de repetição.

Esse é um bom exercício para fixar e aplicar os conceitos aprendidos até este momento, além de ser mais um desafio para você. Tente!



Para saber mais

Existem diversos métodos para ordenação ou classificação – interna ou externa – de dados e arquivos. Neste capítulo, usando o tipo de classificação interna, você aprendeu o funcionamento do método de ordenação por seleção direta aplicado a uma estrutura de dados do tipo vetor. Entretanto, esse mesmo método pode ser aplicado à estrutura de dados *árvore binária* que você estudará no Capítulo 11. Outro método interessante é o *heapsort* (que usa o princípio da ordenação por seleção), que você pode utilizar para classificação de árvores binárias. Sugerimos que você leia sobre isso nos tópicos contidos a partir da página 110 do livro *Algoritmos: teoria e prática* (Cormen *et al.*).

Lembre-se

Nos torneios de duplas, em que o primeiro colocado é selecionado em várias etapas (oitavas, quartas, semifinais e finais), você encontra o princípio do *heapsort*.

Bons estudos!



Navegar é preciso

Atualmente, o ensino de algoritmos e estruturas de dados segue uma abordagem construtivista apoiada por objetos de aprendizagem, com destaque para *softwares* de animação gráfica, nos quais o aprendiz pode observar a execução do algoritmo de forma até certo ponto lúdica. Sugerimos que você acesse os sites a seguir para saber mais sobre algumas iniciativas na construção desse tipo de software, que vêm sendo desenvolvidas por professores, alunos e pesquisadores.

<http://www.ic.unicamp.br/~rezende/garcia.htm>

<http://monografias.cic.unb.br/dspace/bitstream/123456789/315/1/monografia.pdf>

<http://www.br-ie.org/pub/index.php/sbie/article/download/1538/1303>

No site <http://www.inf.ufrgs.br/~bsguedes/simulador/>, você encontrará um simulador didático de testes de algoritmos de ordenação, no qual poderá observar graficamente o desempenho dos vários métodos de ordenação. Nele você terá acesso também a um arquivo PDF contendo a documentação desse trabalho, desenvolvido por alunos do Instituto de Informática da Universidade Federal do Rio Grande do Sul.

No site <http://wikipedia.artudi.org/Bubble%20Sort.php> você encontrará um simulador do método *BubbleSort* e poderá acompanhar, passo a passo, a execução desse método, escolhendo uma quantidade de números para ordenação.

No site <http://www.youtube.com/watch?v=eHeoPABS284>, você encontrará um vídeo de simulação do método *QuickSort* na forma de jogo de tabuleiro. Você poderá acompanhar essa simulação com o algoritmo do método *QuickSort*, estudado neste capítulo.

O site <http://www.youtube.com/watch?v=t8g-iYGHpEA> traz um vídeo em que um programador de computador associou a frequência sonora dos caracteres sendo ordenados e, com isso, conseguiu produzir “o som dos algoritmos de ordenação”. Ele associou o som à representação gráfica de um vetor sendo ordenado, conforme alguns métodos de ordenação. Muito interessante!

Para encerrar esta seção, sugerimos acesso ao site <http://www.youtube.com/watch?v=YKIDz1J3TSw>, em que temos um vídeo interessante demonstrando, de forma lúdica, os métodos de ordenação apresentados neste capítulo, além de outros que não foram objeto de nossos estudos.

Exercícios

1. Considerando a estrutura de dados a seguir, formada pelas colunas “Nome do paciente” e “Idade (em anos)”, e usando o “método da bolha”, classifique as idades em ordem crescente e, em seguida, mostre os nomes dos pacientes e suas idades.

Nome do Paciente	Idade (em anos)
Pedro	32
Maria	45
João	17
Ana	21
Beatriz	2
Luiza	4
Alberto	29
Paulo	56
Roberta	27
Diego	14

2. Considere uma estrutura de dados formada pelas colunas “Nome do aluno” e “Média final”. Usando o método de ordenação *QuickSort*, classifique os alunos pelo seu nome em ordem crescente e, em seguida, mostre o nome de todos os alunos e suas médias finais.
3. Considerando a estrutura de dados a seguir, ordene-a por meio do método da seleção direta - *SelectionSort*. Mostre o conteúdo da estrutura antes e depois da ordenação.

CB DB AA BA CA DA BB AB

Todo programa é um desafio para nossa criatividade!

Vamos lá, mostre que você é criativo!

Lembre-se

A estrutura de dados do Exercício 3 armazena dados do tipo *string*, e esse tipo de dado é tratado de forma diferente por determinadas linguagens. Portanto, devemos observar como essa estrutura será tratada (comparando conteúdo de variáveis do tipo *string*) pelas linguagens que vamos utilizar nesta seção.

Vamos em frente!

Glossário

Yotabyte: unidade de medida em computação que equivale a 10 elevado a 24 byte (10^{24}). O nome *Yottabyte* é derivado da nona letra do alfabeto grego: I ou i (iota).

1 YB = 1.208.925.819.614.629.174.706.176 bytes = 1.000^8 bytes = 10^{24} bytes

Sort: (1) ordenar itens em grupos de acordo com critérios especificados.
(2) ordenar um conjunto de itens de acordo com chaves que são utilizadas como uma base para determinar a sequência dos itens.

BubbleSort: procedimento de troca de itens em que a sequência de verificação de pares de itens é revertida sempre que uma troca é feita. Sinônimo de *borbulhar* ou *peneirar*.

HeapSort: algoritmo de ordenação generalista que aplica o método de ordenação por seleção. Foi desenvolvido em 1964, por Robert W. Floyd e J. W. J. Williams.

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- _____. *Introduction to Algorithms: A Creative Approach*. Boston: Addison Wesley, 1989.
- LAFORE, R. . *Estrutura de dados & algoritmos em Java* Trad. Eveline Vieira Machado. Rio de Janeiro: Ciência Moderna, 2004.
- PIVA, JR., D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Campus, 2012.
- VELOSO, P. et al. *Estruturas de dados*. Rio de Janeiro: Campus, 1983.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.



QUE VEM DEPOIS

Agora que você entendeu a importância da ordenação de dados e informações e está familiarizado com alguns métodos de ordenação, prepare-se para o próximo assunto: alocação dinâmica de dados e estruturas de dados em memória, em tempo de execução do programa.

O que você aprenderá no Capítulo 6 certamente poderá ser utilizado para a otimização dos algoritmos de ordenação que aprendeu neste capítulo.

Vamos em frente e bons estudos!

Alocação dinâmica

“A mente que se abre para uma nova ideia jamais volta ao seu tamanho original.”

ALBERT EINSTEIN

É fundamental ser capaz de armazenar novas informações, que, muitas vezes, nem éramos capazes de supor que quiséssemos guardar.

OBJETIVOS DO CAPÍTULO

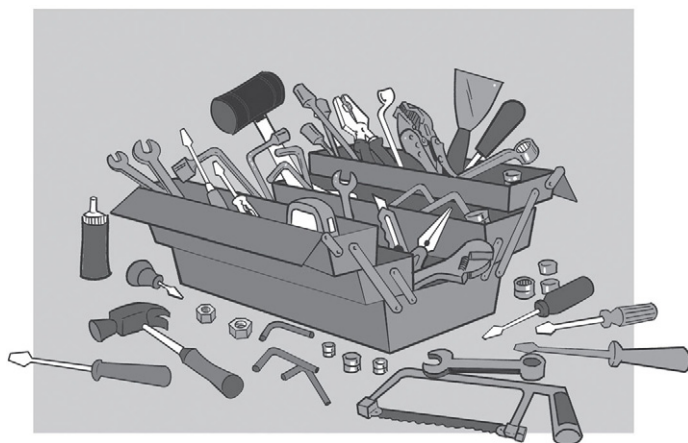
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- identificar situações nas quais é importante utilizar a alocação dinâmica;
- alocar e liberar memória;
- criar aplicações com alocação dinâmica de memória.



Para começar

O que devemos fazer se tivermos de armazenar mais informações do que tínhamos previsto?



Algumas vezes, já sabemos a quantidade de informações que vamos armazenar quando implementamos nosso sistema. Podemos ter, por exemplo, 10 números inteiros ou 500 registros com informações sobre nossos amigos. Outras vezes, porém, a quantidade de informações a serem armazenadas não pode ser prevista *a priori*, ou seja, antes que nosso sistema seja usado na prática. Se não tivermos previsto um número suficiente de variáveis ou um tamanho suficiente para nossas estruturas de dados (como vetores ou matrizes), não teremos espaço para armazenar todas as informações.

Imagine que nosso cliente ou o usuário do nosso sistema não possa cadastrar um novo produto ou executar uma venda porque nós não previmos um tamanho suficiente para nossas estruturas de dados. Ele não ficaria nada feliz, não é?

De outro lado, imagine que, para não correr esse risco, declaramos sempre nossas estruturas de dados com tamanhos muito grandes. Nesse caso, elas necessitariam de muita memória, possivelmente mais memória do que a disponível.

Situação complicada, não? Felizmente existem as *estruturas de dados dinâmicas*, que são aquelas que construímos *sob demanda*, isto é, ocupamos a memória conforme ela vai sendo necessária para armazenar os dados. Entre as estruturas de dados dinâmicas podemos citar as *listas ligadas*, as *filas*, as *pilhas*, as *árvores* e os *grafos*. Algumas dessas estruturas podem ser implementadas também com estruturas de dados convencionais, que chamaremos de *estáticas*.

Neste capítulo, vamos aprender a reservar (alocar) memória dinamicamente (sob demanda) para construir essas estruturas de dados.

Vamos lá!

Atenção

Nas estruturas de dados estáticas, a memória alocada tem tamanho predefinido. Mesmo que não armazenemos nada, o espaço reservado não poderá ser utilizado para armazenar outras informações. E, se precisarmos de mais espaço, isso não será possível; para tanto, precisaríamos reprogramar e recompilar o programa.

Já nas estruturas de dados dinâmicas, a memória é alocada conforme vai sendo necessária para armazenar os dados, ou seja, é alocada “em tempo de execução”.



Uma questão importante é saber se devemos usar estruturas de dados estáticas ou dinâmicas. Quando sabemos antecipadamente quantas variáveis ou quantos valores vamos armazenar, utilizamos *variáveis estáticas*. Podemos também utilizá-las se soubermos que a quantidade máxima de dados que teremos de armazenar será pequena (no máximo 100 valores, por exemplo). De outro lado, se não soubermos a quantidade de dados e se essa quantidade for grande, geralmente usamos *variáveis dinâmicas*.

Propomos, a seguir, algumas situações para você identificar quantas variáveis serão necessárias e qual o seu tamanho (por exemplo: duas variáveis inteiras, ou um vetor de inteiros com 20 posições). Quando não for possível saber a quantidade de dados, responda “variáveis dinâmicas”.

1. Armazenar os tamanhos dos lados de um triângulo.
2. Identificar o maior entre 10 números reais.
3. Armazenar as notas das provas de Estruturas de Dados dos alunos da sua turma.
4. Calcular e imprimir a média das notas de cada aluno.
5. Ler e armazenar valores inteiros até que o valor lido seja -1.
6. Obter a temperatura (de um sensor) a cada hora e calcular a maior e a menor temperatura do dia.
7. Guardar a maior e a menor temperatura de cada dia.
8. Armazenar o nome e um e-mail de no máximo 1.000 amigos.
9. Armazenar o nome e um e-mail de todos os amigos.

Conseguiu identificar? Ficou com dúvida em muitos itens?

Vamos aprender um pouco mais para eliminar as dúvidas e utilizar a alocação dinâmica de memória.



Conhecendo a teoria para programar

As variáveis que temos usado até aqui (estáticas) são armazenadas na área de variáveis globais ou na pilha (*stack*) do processo (nosso programa, quando está sendo executado), sendo identificadas e acessadas por seus nomes. Assim, podemos ter duas variáveis inteiras, *i* e *j*, e uma *string* chamada *nome* com 20 caracteres, por exemplo.

As variáveis dinâmicas, de outro lado, são armazenadas no *heap* do processo. Como não sabemos quanta informação vamos armazenar, não podemos dar nomes a ela. Como poderemos acessá-las, então? Nesse caso, o acesso é feito por meio de ponteiros, ou seja, de seus endereços de memória.

Lembre-se

A maioria das variáveis armazena valores (inteiros, caracteres, números reais, ...). Um ponteiro também é uma variável, mas, em vez de armazenar o valor de um dado, armazena um endereço de memória.

No Capítulo 14 do livro *Algoritmos e programação de computadores*, também de nossa autoria, você encontrará detalhes sobre os ponteiros.

Para obter espaço na memória enquanto o programa está sendo executado (chamamos a isso de *alocação de memória*), usamos principalmente o comando *malloc*, que está presente na biblioteca *stdlib.h*. O comando *malloc* possui apenas um parâmetro, que define quantos *bytes* serão alocados. Assim, *malloc(4)*, no Código 6.1, aloca 4 *bytes*, e *malloc(20)* aloca 20 *bytes* na área de *heap*.

Além disso, o comando *malloc* retorna o endereço do primeiro *byte* alocado. Por que ele faz isso? Para podermos usar esse espaço de memória por meio de seu endereço, já que esse espaço não tem um nome (como *i*, *j* ou *nome*, por exemplo). Como o comando *malloc* não sabe que tipo de informação nós vamos armazenar nesse espaço de memória, ele retorna o endereço (ponteiro) para um *void*. Nós devemos informar qual tipo de informação vamos armazenar, por meio de um *cast*.



Conceito

Um *cast* nada mais é que uma conversão de tipos. O comando *malloc* retorna um *void**, isto é, um ponteiro (ou endereço) para *void*. Como não é isso que vamos armazenar naquele espaço de memória, devemos indicar o que será. Para armazenar um valor inteiro naquele endereço, faremos um *cast* para *int**. Se formos armazenar um número real, faremos um *cast* para *float**, e assim por diante.

Vamos agora alocar espaço para armazenar um número inteiro e uma *string* com 20 caracteres. Em seguida, vamos ocupar esse espaço de memória por meio de seus endereços, armazenados em ponteiros (ponteiro *p* para armazenar o endereço do número inteiro, e ponteiro *s* para armazenar o endereço do [primeiro] caractere).

Código 6.1

```
int *p;                // ponteiro para inteiro
char *s;               // ponteiro para caracter
p = (int *) malloc(4); // supomos que um número inteiro ocupa 4 bytes.
*p = 7;
s = (char *) malloc(20); // supomos que cada caractere ocupa 1 byte.
printf("Digite a string:\n");
scanf("%s",s);
printf("Conteudo de p=%d Conteudo de s=%s\n",*p,s);
```

Bem, como estamos desenvolvendo programas cada vez mais complexos, podemos querer executá-los em diferentes ambientes (com arquiteturas, sistemas operacionais ou compiladores diferentes).

E se eles usassem tamanhos diferentes para os tipos de dados, como 2 bytes, 4 bytes ou 8 bytes, por exemplo, para armazenar um número inteiro? Nesse caso, teríamos que alterar o programa para executar na nova instalação.

Para evitar esse problema e dar *portabilidade* aos nossos programas, usaremos a função *sizeof*, que retorna o tamanho do parâmetro em *bytes*. Assim, se mudarmos de ambiente de desenvolvimento, poderemos recompilar o mesmo programa sem precisar alterá-lo. Ele é mostrado no Código 6.2.

```
int *p; // ponteiro para inteiro
char *s; // ponteiro para caracter

p = (int *) malloc(sizeof(int)); // alocamos espaço para um inteiro
*p = 7;

s = (char *) malloc(20*sizeof(char)); // alocamos espaço para 20 caracteres
printf("Digite a string:\n");
scanf("%s",s);

printf("Conteudo de p=%d Conteudo de s=%s\n",*p,s);
```

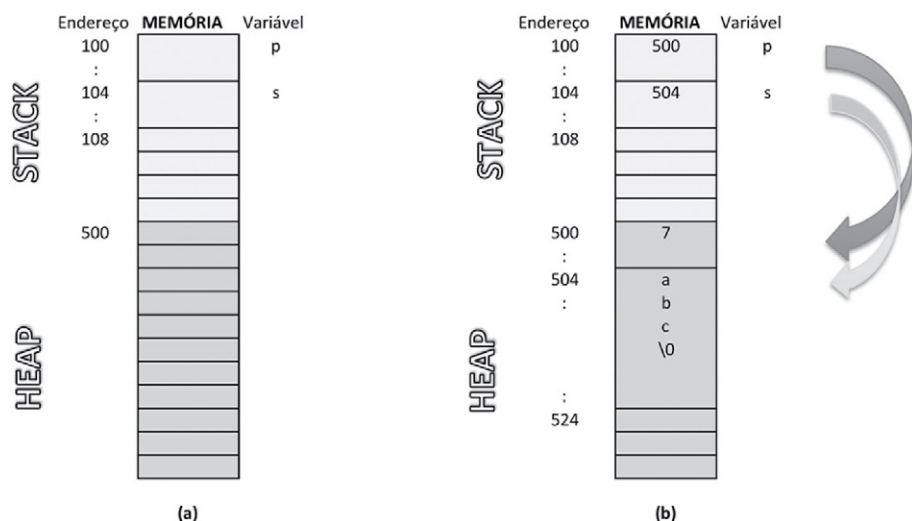


FIGURA 6.1: Alocação de um inteiro e uma *string* de 20 caracteres na memória.

O primeiro comando *malloc* vai alocar 4 bytes (*sizeof(int)*) na área de *heap*, que é onde ficam as variáveis dinâmicas. Informamos ao comando *malloc* que vamos armazenar um número inteiro naquele espaço de memória por meio do *cast(int *)*. Além disso, o comando *malloc* retorna o endereço do primeiro byte alocado (500), que armazenamos na variável *p* (lembre-se de que *p* é um ponteiro para inteiro, ou seja, ele armazenará o endereço de um número inteiro). Assim, vemos, na Figura 6.2b, que a variável *p* recebeu o valor 500. Ela é chamada de ponteiro porque os dados serão armazenados no endereço 500, e não no endereço 100, onde a variável *p* está. Para acessar os dados a partir de *p*, temos que seguir uma seta (ponteiro) até o endereço 500. Assim, quando escrevemos **p* estamos seguindo essa seta e acessando o conteúdo do endereço 500. Dessa forma, o comando **p = 7* armazena o valor 7 no endereço 500. Por enquanto, deve parecer estranho fazer isso, mas esse mecanismo vai ser muito útil na construção das próximas estruturas de dados.

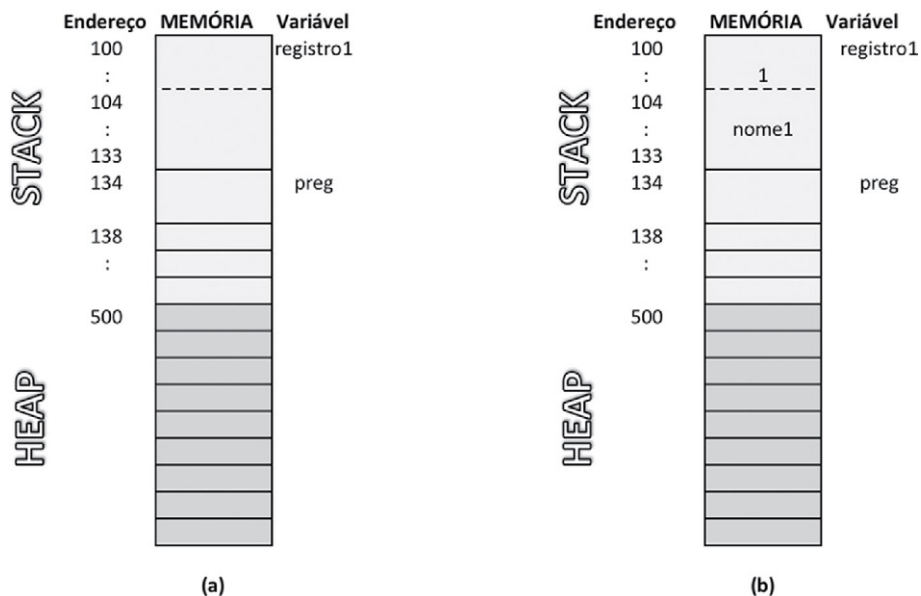


FIGURA 6.2: Registro e ponteiro na pilha (stack).

Dica

Quando não há memória disponível no *heap*, o comando *malloc* retorna o endereço especial *NULL* (zero). Se atribuirmos o valor de retorno do comando *malloc* a um ponteiro *p*, por exemplo, podemos testar se *p* é *NULL* para sabermos se a área de *heap* está cheia.

Além disso, como os compiladores C associam o valor zero ao *booleano* FALSE e qualquer valor diferente de zero ao *booleano* TRUE, se o valor de *p* for *NULL* (FALSE), não havia memória disponível; se *p* for TRUE, a memória foi alocada com sucesso.



A execução do segundo comando *malloc* é similar. Ele vai alocar 20 bytes na área de *heap*, a partir do endereço 504. Informamos por meio do *cast(char *)* que nesse espaço de memória serão armazenados caracteres. O comando *malloc* retorna o endereço do primeiro byte alocado (504), que atribuiremos à variável *s*. Agora uma diferença: para acessar o endereço 504 por meio do ponteiro *s* não utilizamos o operador* de ponteiros (*s), mas somente o nome da variável (*s*). Por quê? Tente descobrir isso! Dica: lembre-se do funcionamento das *strings*, e de como *strings*, vetores, matrizes e registros são passados como parâmetros para funções. Um bom livro sobre algoritmos e programação poderá ajudá-lo.

Papo técnico

Sabemos que um ponteiro armazena um endereço de memória. Os ponteiros *p* e *s* da [Figura 6.1](#) têm 4 bytes cada, ou seja, 32 *bits*. Isso significa que eles podem armazenar 2^{32} valores diferentes. Como $2^{32} = 4\text{G}$, um ponteiro com 4 bytes pode endereçar até 4G endereços de memória. Se cada endereço tiver 1B (1 byte), ele poderá endereçar uma memória de até 4GB.

Nos ambientes com 64 *bits*, os ponteiros têm até 64 *bits*, ou seja, 8 bytes. Eles podem armazenar até 2^{64} valores, ou seja, 16 EB (*exabytes*), o que equivale a 16 milhões de *terabytes*!



Nossas próximas estruturas de dados – listas, filas, pilhas e árvores – usarão registros quando implementadas com alocação dinâmica. Vamos ver um exemplo que mostra a diferença no uso de registros quando estiverem na pilha e no *heap*. Conforme for lendo o código, tente imaginar o que ocorrerá na memória do computador.

Código 6.3

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//Definição do registro
typedef struct tipo_registro registro;

struct tipo_registro {
    int codigo;
    char nome[30];
};

int main()
{
    //Declaração das variáveis
    registro registro1; // registro convencional
    registro *preg;      // ponteiro para um registro

    // Atribuição de valores para o registro convencional
    registro1.codigo = 1;
    strcpy(registro1.nome, "nome1");

    // Alocação de um registro na área de heap
    preg = (registro *) malloc (sizeof(registro));
    if ( !preg ) printf("Não há memória disponível\n");
    else{
        //Atribuição de valores para o registro alocado
        preg->codigo = 2;
        strcpy(preg->nome, "nome2");

        // Liberação da memória alocada
        free (preg);
    }
}
```


No Código 6.3, vemos a declaração do novo tipo, chamado *registro*, que é uma *struct r* (a *struct r*, definida a seguir, possui um campo inteiro e um campo *string*). Essa declaração não gera código executável, e, assim, não ocupa memória do processo. As variáveis *registro1* e *preg* são variáveis locais (da função *main*), e, assim, são alocadas na pilha do processo. A Figura 6.2a mostra a variável *registro1* alocada no endereço 100 de memória. Ela ocupa 34 bytes, sendo 4 bytes para o atributo *codigo* (inteiro) e 30 bytes para o atributo *nome* (30 caracteres). Ocupa, assim, as posições de memória de 100 a 133. A variável *preg* está alocada no endereço 134 e ocupa 4 bytes (ponteiro).

A seguir, o programa mostra a atribuição de valores para o *registro1*. Como sabemos, o acesso aos atributos do registro é feito colocando-se um ponto entre o nome da variável e o nome do atributo (por exemplo, *registro1.codigo*). O atributo *codigo* recebe o valor 1 e o atributo *nome* recebe o valor “nome1” (lembre-se de que a atribuição de valores é realizada por meio do comando *strcpy* em *strings*). A Figura 6.2b mostra a memória após as atribuições de valores.

Depois, o programa aloca memória para armazenar um registro. O comando *malloc* solicita um número de bytes equivalente a *sizeof(registro)*, ou seja, 34 bytes no nosso exemplo. Os bytes solicitados pelo comando *malloc* são alocados na área de *heap*, nos endereços 600 a 633. Como o comando *malloc* retorna o endereço do primeiro byte alocado, que é 600, esse valor é guardado no ponteiro *preg*, como mostra a Figura 6.3a. Se não houvesse memória suficiente no *heap*, o comando *malloc* retornaria o ponteiro especial *NULL* (endereço zero), e o programa imprimiria uma mensagem de erro.

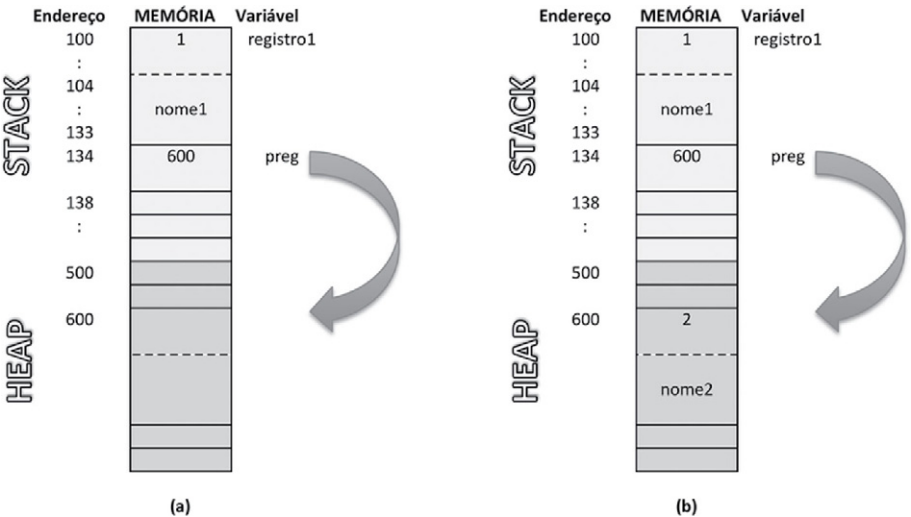


FIGURA 6.3: Alocação de um registro no heap.



Atenção

Uma sequência de comandos *malloc* não vai necessariamente alocar posições consecutivas de memória. Ela poderia alocar memória primeiro na posição 600, depois na posição 1200 e então na posição 700, por exemplo.

Então, o programa armazena valores no *heap*, no espaço que acabamos de alocar. Note que não podemos identificar os atributos do registro, como fizemos com o *registro1* (e utilizar, por exemplo, *preg.codigo*), porque ele não está nas posições de memória da variável *preg*. Para acessar o registro alocado, temos de seguir uma seta (ou um ponteiro) até o endereço de memória contido em *preg*, ou seja, seguir uma seta até o endereço 600. É lá que vamos acessar os atributos do registro. Para fazer isso, em vez de identificarmos os atributos com “ponto”, vamos identificá-los com uma “seta”, formada pelo caractere *hifen* seguido pelo caractere *maior*, ou seja, - >. Assim, o programa atribui o valor 2 ao atributo *codigo* e o valor *nome2* ao atributo *nome*. A [Figura 6.3b](#) mostra a memória após essas atribuições.

Por fim, o programa mostra o uso do comando *free*, que libera memória do *heap* a fim de que possa ser utilizada para armazenar outras informações. No exemplo, o comando *free(preg)* faz com que o espaço de memória iniciado no endereço 600 (valor de *preg*) seja liberado.

Dica

Tome muito cuidado com o uso dos ponteiros. Alguns erros comuns:

1. acessar um atributo por meio de um ponteiro cujo valor é zero (NULL). Nesse caso, ao tentar acessar o endereço zero de memória, o programa vai abortar, isto é, interromper sua execução;
2. inicie sempre os ponteiros, mas não por meio do comando *malloc*. O comando *malloc* deve ser usado exclusivamente quando tivermos uma nova informação para armazenar, e então solicitaremos mais memória;
3. ao executar o comando *free*, a área de memória será liberada, mas o ponteiro continuará guardando o mesmo valor. Se tentarmos acessá-lo novamente, não teremos como saber qual será o valor armazenado, pois os endereços liberados já podem estar ocupados com outras informações.



Bem, mas aí surge uma questão: se precisássemos ter uma variável do tipo ponteiro para cada registro, teríamos de saber a quantidade de registros a serem armazenados quando estivéssemos programando. Nosso próximo passo, portanto, é conhecer a estratégia que usaremos para acessar um registro a partir de outro registro, e não a partir de uma variável. O segredo é que nossos registros vão armazenar o endereço de um (ou mais) registros, e assim seguiremos uma cadeia de endereços. Parece complicado, não é? Vamos, então, ver um exemplo para entender melhor esse processo.

Em primeiro lugar, vamos redefinir nosso registro da seguinte forma:

```
typedef struct tipo_registro registro;  
struct tipo_registro {  
    int codigo;  
    char nome[30];  
    struct tipo_registro *prox;  
};
```

Observamos que, além das informações que queremos armazenar sobre amigos, clientes ou produtos (representadas no exemplo pelos campos *codigo* e *nome*), inserimos outro campo, chamado *prox* (que armazenará o endereço do próximo registro). Como esse campo armazenará o endereço de um registro, será um ponteiro para o registro, isto é, seu tipo será *registro**.

No Código 6.4, temos dois ponteiros para registros: *p1* e *p2*. Como os ponteiros são apenas endereços, precisamos alocar memória para cada um e, assim, armazenar informações neles. Mas desta vez vamos inserir o endereço do segundo registro dentro do primeiro (no campo *prox*). Vamos ver como isso vai funcionar.

Código 6.4

```
#include <stdio.h>

#include <string.h>

#include <stdlib.h>

typedef struct tipo_registro registro;

struct tipo_registro {    //Definição do registro

    int codigo;

    char nome[30];

    struct tipo_registro *prox;

};

int main()

{

    registro *p1, *p2;    // ponteiros para os registros

    p1=(registro *) malloc (sizeof(registro);    //Alocação dos registro na área de heap

    p2=(registro *) malloc (sizeof(registro);

    if ((!p1) || (!p2)) printf("Não há memória disponível\n");

        else{    //Atribuição de valores para o primeiro registro

            p1->codigo = 1;

            strcpy(p1->nome, "nome1");

            p1->prox = p2;

            //Atribuição de valores para o segundo registro

            p2->codigo = 1;

            strcpy(p2->nome, "nome2");

            p2->prox = NULL;

            // Exemplos de acesso aos registros

            printf("Codigo de p1: %d\n", p1->codigo);

            printf("Nome de p2: %s\n", p2->nome);

            printf("Endereco contido no campo prox de p1: %d\n", p1->prox);

            printf("Endereco contido no campo prox de p2: %d\n", p2->prox);

            printf("Codigo de p2: %d\n", p2->codigo);

            printf("Codigo de p2: %d\n", p1->prox->codigo);

        }

}
```

Neste programa, temos apenas duas variáveis na função *main*, *p1* e *p2*. Como sabemos, elas são alocadas na pilha do processo. A [Figura 6.4a](#) mostra *p1* alocada no endereço 100 e *p2* no endereço 104 de memória.

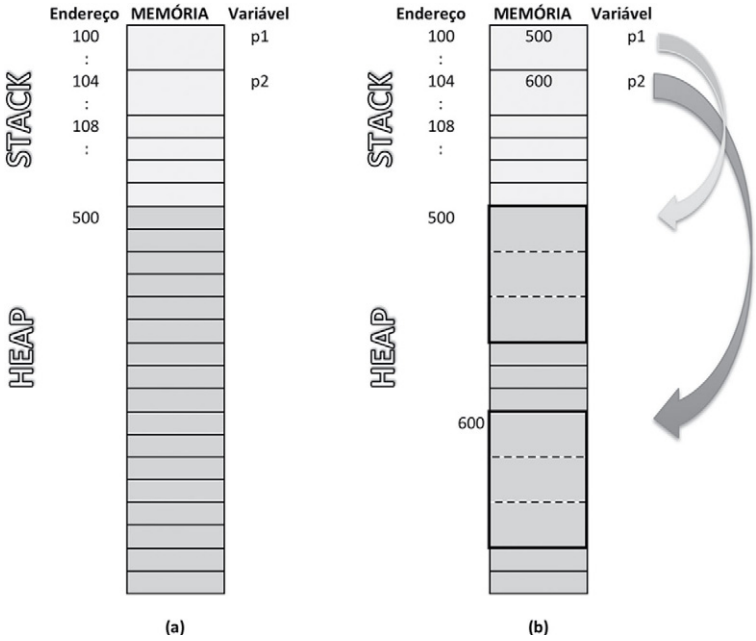


FIGURA 6.4: Alocação de dois registros.

Na execução do primeiro comando *malloc*, uma área de memória suficiente para armazenar um *registro* é alocada, e seu endereço (500) é atribuído à variável *p1*. De forma similar, o segundo comando *malloc* aloca espaço a partir do endereço 600, e esse valor é atribuído à variável *p2*, como mostra a [Figura 6.4b](#).

Os registros alocados nos endereços 500 e 600 foram redesenhados na horizontal, como podemos ver a seguir. Uma vez que o ponteiro *p1* está armazenando 500, que é o endereço do primeiro registro, colocamos o ponteiro *p1* apontando para esse registro. Fizemos o mesmo para *p2*, que contém 600 (endereço do segundo registro).



Na sequência do Código 6.4, atribuímos valores aos atributos de *p1* e *p2*, acessando-os por meio de setas, como fizemos anteriormente. O resultado pode ser visto na [Figura 6.5](#).

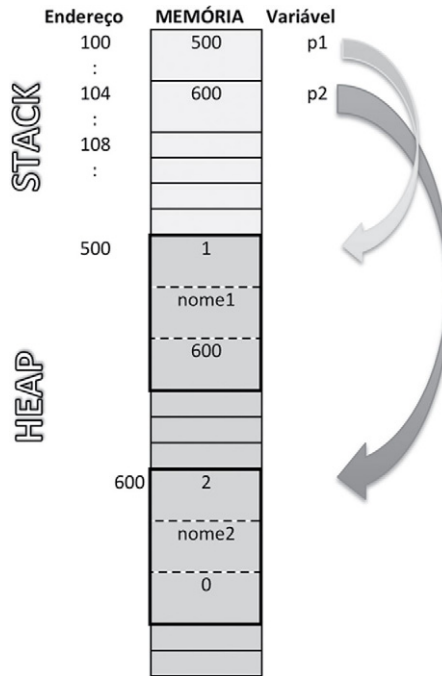


FIGURA 6.5: Atribuição de valores aos dois registros.

Observe que, no campo *prox* do primeiro registro, inserimos o valor 600 (valor de *p2*) para indicar que, após o primeiro registro, está o segundo registro. No campo *prox* do segundo registro inserimos *NULL* para indicar que não há nenhum registro subsequente. Na nossa segunda forma de representação, os registros armazenados na memória teriam a seguinte estrutura:



O primeiro registro, armazenado no endereço 500, contém três campos: o campo *codigo* armazena o valor 1, o campo *nome* armazena o valor

“nome1” e o campo *prox* armazena o valor 600. Como 600 é o endereço do segundo registro, nesse campo desenhamos uma seta para o segundo registro, indicando que ele “aponta” para o registro que está no endereço 600. No campo *prox* do segundo registro está armazenado o endereço especial *NULL* (zero), indicando que não há nenhum registro à frente. Nas próximas figuras não vamos mostrar os valores nos campos de endereços (campo *prox*), apenas as setas indicando para quais registros eles apontam. O ponteiro *NULL* será representado através do símbolo “\”.

Há mais um detalhe: você percebeu que é possível acessar o conteúdo do segundo registro sem utilizar o ponteiro *p2*? Observe novamente os dois últimos comandos do programa. O primeiro comando acessa o campo código do segundo registro pelo ponteiro *p2*, cujo valor é 600. Assim, para avaliar a expressão *p2->código*, o compilador¹ verifica primeiro o conteúdo de *p2* (que é 600). Como existe uma seta (->), ele sabe que deve seguir para o endereço 600, e é lá que ele vai procurar o campo *código*. O campo *código* do endereço 600 contém o valor 2, e é esse valor que será impresso.

E a execução do segundo comando? Para avaliar a expressão *p1->prox->código*, o compilador verifica o conteúdo de *p1* (que é 500). Então, ele segue para o endereço 500, e lá vai procurar o campo *prox*. No campo *prox* do registro que está armazenado no endereço 500, ele acha o valor 600. De novo, ele vê uma seta e sabe que tem que seguir para o endereço 600. Lá, ele procura o campo *código*, onde encontra e imprime novamente o valor 2.

Assim, vimos que é possível acessar o conteúdo do segundo registro a partir do primeiro registro. Vamos exercitar um pouco e passar às listas ligadas, em que armazenaremos quantos registros quisermos, podendo acessá-los a partir de um único ponteiro (que conterà o endereço do primeiro registro).



Vamos programar

Java

Java faz alocação de memória automaticamente, por isso não é necessário solicitá-la de forma explícita por meio de um comando como *malloc*. Por isso, os Códigos 6.1 e 6.2 foram omitidos. A seguir, você encontrará exemplos para armazenar dados em registros, acessá-los e ligá-los.

¹Mais especificamente, o compilador gera o código que fará isso.

Código 6.3

```

public class Registro {
    private int codigo;
    private String nome;
    private Registro prox;

    public Registro() {
    }

    public int getCodigo() {
        return this.codigo;
    }

    public String getNome() {
        return this.nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }
}

##MAIN##
public class JavaApplication2 {
    public static void main(String[] args) {
        Registro registro1 = new Registro();
        registro1.setCodigo(1);
        registro1.setNome("nome1");
        Registro registro2 = new Registro();
        registro2.setCodigo(2);
        registro2.setNome("nome2");
        System.out.println("Nome: " + registro1.getNome() + "   " + "Codigo: " +
registro1.getCodigo());
        System.out.println("Nome: " + registro2.getNome() + "   " + "Codigo: " +
registro2.getCodigo());
    }
}

```


Código 6.4

```

public class Registro {

    private int codigo;
    private String nome;
    private Registro prox;

    public Registro(){
    }

    public int getCodigo(){
        return this.codigo;
    }

    public String getNome(){
        return this.nome;
    }

    public void setNome(String nome){
        this.nome = nome;
    }

    public void setCodigo(int codigo){
        this.codigo = codigo;
    }

    public Registro getProx(){
        return this.prox;
    }

    public void setProx(Registro prox){
        this.prox = prox;
    }
}

##MAIN##

public class JavaApplication2 {

    public static void main(String[] args) {

        Registro p1 = new Registro();
        Registro p2 = new Registro();

        p1.setCodigo(1);
        p1.setNome("nome1");
        p2.setProx(p1);
        p2.setCodigo(1);
        p2.setNome("nome2");
        p2.setProx(null);

        System.out.println("Codigo de p1: " + p1.getCodigo());
        System.out.println("Nome de p2: " + p2.getNome());
        System.out.println("Endereco contido no campo prox de p1: " +
System.identityHashCode(p1.getProx()));
        System.out.println("Endereco contido no campo prox de p2: " +
System.identityHashCode(p2.getProx()));
        System.out.println("Codigo de p2: " + p2.getCodigo());
        System.out.println("Codigo de p2: " + p2.getCodigo());
    }
}

```

Python

Assim como Java, Python também faz alocação de memória automaticamente, por isso não é necessário solicitá-la de forma explícita por meio de um comando como *malloc*. Por isso, os Códigos 6.1 e 6.2 foram omitidos. A seguir, você encontrará exemplos para armazenar dados em registros, acessá-los e ligá-los.

Código 6.3

```
#Definição do registro
class Registro :
    #constructor
    def __init__(self, codigo=None, nome=None):
        self.codigo = codigo
        self.nome = nome
    #toString()
    def __str__(self):
        return str('Codigo:%s\nNome:%s' % (self.codigo, self.nome))

#main()
#Declaração das variáveis e alocação de um registro na área de heap
registro1 = Registro()
#Atribuição de valores para o registro convencional e alocado
registro1.codigo=1
registro1.nome="nome1"
registro2 = Registro()
registro2.codigo=2
registro2.nome="nome2"
#toString
print registro1
print registro2
```

(Continua)

Código 6.4

```
#Definição do registro
class RegistroLigado :
    #construtor
    def __init__(self, codigo=None, nome=None, prox=None):
        self.codigo = codigo
        self.nome = nome
        self.prox = prox
    #toString()
    def __str__(self):
        return str('Codigo:%s\nNome:%s' %(self.codigo,self.nome))

#main()
#Alocação dos registros na área de heap
p1 = RegistroLigado()
p2 = RegistroLigado()

#Atribuição de valores para o primeiro registro
p1.codigo=1
p1.nome="nome1"
p1.prox=p2

#Atribuição de valores para o segundo registro
p2.codigo=2
p2.nome="nome2"

#Exemplos de acesso aos registros
print "Codigo de p1: %d" %(p1.codigo)
print "Nome de p2: %s" %(p2.nome)
print "Endereço contido no campo prox de p1:", hex(id(p1.prox))
#Só para confirmar se o endereço está correto
print "Endereço de p2:", hex(id(p2))
print "Endereço contido no campo prox de p2:", hex(id(p2.prox))
print "Codigo de p2: %d" %(p2.codigo)
print "Codigo de p2: %d" %(p1.prox.codigo)
```



Para fixar

1. Implemente um programa que aloque memória para três registros, utilizando um ponteiro para cada registro. Lembre-se de preencher todos os campos dos registros e ligá-los (o endereço do segundo registro deve ser armazenado no primeiro; o endereço do terceiro registro deve ser armazenado). Não se esqueça de inserir NULL no último registro (isso é muito importante, como veremos no Capítulo 7).
2. Em seguida, imprima os dados armazenados. Verifique como acessar as informações somente a partir do ponteiro para o primeiro registro (por exemplo, *p1*). Você também pode acessar as informações do segundo e terceiro registros a partir do ponteiro para o segundo registro. Faça o teste!



Para saber mais

Os capítulos 13 e 14 (Registros e Ponteiros) do livro *Algoritmos e programação de computadores* (Piva *et al.*) apresenta detalhes do uso de registros e ponteiros que podem ser muito úteis no estudo da alocação dinâmica de memória e uso de estruturas dinâmicas.



Navegar é preciso

Há vários materiais que exemplificam de outras formas o uso de alocação dinâmica de memória. Você poderá encontrá-los, por exemplo, em <http://www.inf.puc-rio.br/~inf1007/material/slides/alocacaodinamica.pdf> e em <http://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html>. No primeiro *link* há vários exemplos e figuras que mostram o uso de alocação dinâmica, incluindo seu uso com vetores. O segundo explica em detalhes o funcionamento dos comandos *malloc* e *free*. Nele você vai ficar sabendo por que alocar grandes porções de memória poucas vezes é melhor do que alocar pequenas porções muitas vezes.

Exercício

Verifique como seu compilador/ambiente de desenvolvimento aloca espaço na área de *heap*. Para isso, aloque memória para vários registros e verifique em quais endereços eles foram alocados. Algumas possibilidades que você pode encontrar:

- os endereços são alocados sequencialmente, em ordem crescente;
- os endereços são alocados sequencialmente, em ordem decrescente;
- os endereços não são alocados em sequência.

A seguir, execute um *loop* de alocação de memória até que não haja mais memória disponível no *heap*. Qual é o tamanho do *heap*?

Por fim, libere memória por meio do comando *free* e, em seguida, faça um *loop* de alocação de memória. Os endereços liberados foram utilizados novamente? Demorou para que isso acontecesse?

Você ainda pode verificar se é possível alterar o tamanho do *heap* (muitas vezes, você poderá precisar de mais memória). A maioria dos compiladores permite que você faça isso por meio de diretivas que alteram o esquema de memória.

Glossário

Processo: trata-se de um programa em execução. Assim, você pode ter centenas de programas instalados no seu HD (ou SSD), mas apenas alguns deles executarão em um dado momento. Processos são, portanto, unidades gerenciadas e escalonadas (colocadas para serem executadas) pelo sistema operacional.

Portabilidade: característica dos sistemas que podem ser portados (ou transportados) para ambientes diferentes (computadores, sistemas operacionais etc.) e executados sem necessidade de modificação.

Referência bibliográfica

PIVA, D. *et al.* *Algoritmos e programação de computadores*. Rio de Janeiro: Campus Elsevier, 2012.



QUE VEM DEPOIS

Agora que você já experimentou alocar memória, acessá-la e ligar uma estrutura à outra, chegou a hora de experimentar nossa primeira estrutura de dados dinâmica: as listas ligadas!

Bons estudos!

Listas ligadas lineares

“ É um amor pobre aquele que se pode medir. ”

WILLIAM SHAKESPEARE

Muitas vezes não sabemos o tamanho das coisas verdadeiramente importantes.

OBJETIVOS DO CAPÍTULO

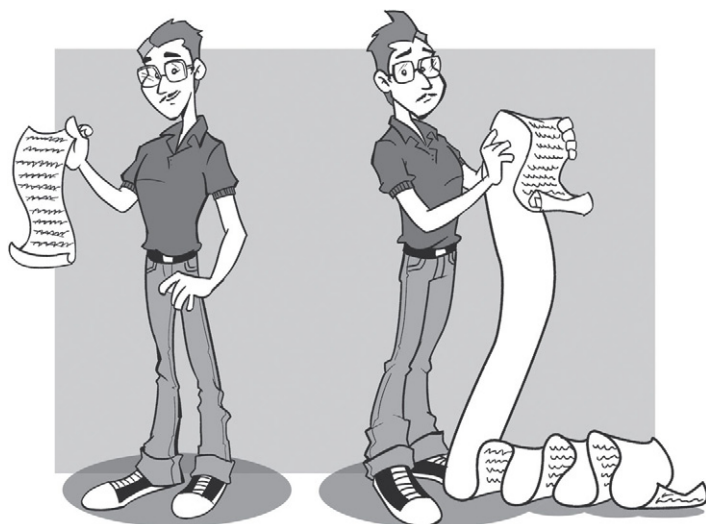
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer as listas ligadas;
- manipular listas, inserindo, removendo e encontrando informações armazenadas;
- utilizar e reconhecer aplicações em que o uso de listas ligadas é adequado.



Para começar

Com o tempo, descobrimos novos itens a serem lembrados e podemos nos esquecer de outros que já realizamos ou que não têm mais importância.



Imagine que você queira fazer uma lista de itens. Poderia ser uma lista de compras, de afazeres, de produtos, de clientes ou uma lista de seus amigos, com nome, endereço, telefone e e-mail de cada um. Você, provavelmente, pensaria em guardar essa lista para ver as informações armazenadas sempre que quisesse. Novas informações poderiam ser guardadas, e informações que não fossem mais úteis seriam removidas da lista. Assim, ela poderia aumentar ou diminuir de tamanho ao longo do tempo, sem que você precisasse reescrevê-la.

Para armazenar uma lista de itens em um computador (ou claro, em um tablet, notebook ou qualquer equipamento móvel), poderíamos usar uma estrutura de dados com estas características:

1. linear ou unidimensional, ou seja, em que pudéssemos buscar as informações sequencialmente, uma após outra;
2. dinâmica, ou seja, que aumentasse e diminuísse sob demanda, conforme o crescimento do sistema;
3. que fosse manipulada em memória (RAM), mais rápida, e depois pudesse ser armazenada em HD ou SSD para que as informações não fossem perdidas.

Um tipo de estrutura de dados com essas características é a *lista ligada linear*, também conhecida somente como *lista ligada*. Ela é a nossa primeira

estrutura de dados dinâmica, e é a partir dela que vamos aprender a manipular os dados de forma cada vez mais flexível, dinâmica e eficiente. Ao trabalho!



Conhecendo a teoria para programar

A *lista ligada linear* é uma estrutura de dados unidimensional e dinâmica formada por elementos denominados *nós*. Cada nó é, usualmente, implementado como um registro, como veremos mais adiante. Por serem dinâmicos, os nós ou dados da lista são armazenados na área de *heap* do processo, sob demanda. Para acessá-los é necessário saber pelo menos o endereço do primeiro nó, que, muitas vezes, é chamado de *cabeça* da lista. Esse primeiro endereço deve ser armazenado por uma variável. As variáveis, como sabemos, são armazenadas na *pilha* do processo.

Sua estrutura e seu funcionamento lembram os dos vetores, que também são estruturas de dados lineares. No entanto, existem algumas diferenças importantes:

1. os elementos dos vetores podem ser acessados diretamente, por meio de seus índices (como, por exemplo, `vet[5]`). Nas listas ligadas, os nós são acessados sequencialmente, um após o outro, pelos ponteiros;
2. geralmente, os vetores são alocados antes da execução do processo. Assim, todo o espaço previsto para o vetor já é alocado e fica reservado para seus dados. Se armazenarmos poucos dados, o espaço reservado será desperdiçado. Nos vetores não é possível armazenar mais dados do que o previsto em sua declaração inicial. Já nas listas ligadas, o espaço em memória é alocado conforme necessário.

A seguir, vemos uma ilustração de uma lista linear com três nós. Os nós possuem uma estrutura conhecida:

```
struct tipo_no {  
    int info;  
    struct tipo_no *prox;  
};
```

Nessa estrutura definimos um tipo chamado *nó*. Cada nó é uma *struct tipo_no*, que é composta por um campo *info* inteiro e um campo *prox*, que conterá o endereço de outro registro. Para simplificar nossa explicação, a

partir de agora vamos trabalhar apenas com um campo de informação (*info*), mas sabemos que podemos acrescentar outros campos, como endereço, telefone, e-mail etc., dependendo do sistema que formos desenvolver. Faríamos como no Capítulo 6, onde também usamos um campo *nome*.

Os três nós da nossa ilustração contêm os valores 1, 2 e 3, respectivamente. Eles estão alocados nos endereços de memória 500, 508 e 516. Para acessar o segundo nó, inserimos seu endereço no campo *prox* do primeiro nó. Da mesma forma, para acessar o terceiro nó, inserimos o endereço 516 no campo *prox* do segundo nó. Como não há nenhum nó após o terceiro, inserimos o ponteiro *NULL* (zero) no campo *prox* do terceiro nó. E para acessar o primeiro nó? Precisaremos de uma variável para armazenar seu endereço. O nome dessa variável na ilustração é *inicio*.



Usaremos também uma ilustração simplificada para facilitar nossa visualização da lista. Não escreveremos mais os endereços de memória, pois, para programar, não importa em quais endereços os nós foram alocados. Por meio das setas saberemos, por exemplo, que o endereço do segundo nó está armazenado no campo *prox* do primeiro nó e que a variável *inicio* armazena o endereço do primeiro nó. Vejamos a mesma lista, agora nessa forma simplificada:



Mão na massa, vamos construir nossa lista! Em vez de construí-la na função principal (*main*), vamos fazer isso por meio da implementação de funções, para dar maior modularidade ao nosso programa. Para evitar erros, vamos usar o mesmo raciocínio no desenvolvimento de funções recursivas: primeiro resolver o caso mais simples (por exemplo, quando a lista for vazia) e, depois, resolver o caso geral (lista com nós). Vamos também declarar e inicializar uma variável na função principal, que vai armazenar o endereço do primeiro nó:

```
no *inicio = NULL; // o ponteiro inicio NULL indica lista vazia.
```



Atenção

Todas as declarações e funções desenvolvidas neste capítulo foram agrupadas e geraram um programa de manipulação de listas (Código 7.1).

Inserção de um nó no início da lista

Para inserir um novo nó com valor n no início da lista, vamos primeiro alocar espaço para o novo elemento e, depois, ligá-lo à lista. Há várias formas de fazer isso, mas vamos dividir o algoritmo no caso mais simples (lista vazia) e no caso geral (lista não vazia). Caso a lista esteja vazia, o novo nó passará a ser o único elemento da lista; senão, vamos inseri-lo como primeiro elemento. O nó que estava no início passará a ser o segundo elemento, e assim por diante, mas não será necessário deslocar os nós fisicamente. Vamos ver como funciona:

```
void insere_inicio (int n, no **inicio)
{
    no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
    if (aux)                               // conseguiu alocar espaço
    {
        aux->info = n;
        if (!(*inicio))                  // lista vazia - é o mesmo que if (*inicio==NULL)
        {
            (*inicio) = aux;
            (*inicio)->prox = NULL;
        }
        else                             // lista não vazia
        {
            aux->prox = (*inicio); // liga o novo nó à lista
            (*inicio) = aux;       // o início da lista passa a ser o novo nó
        }
    }
    else printf ("Heap overflow\n");
}
```

Atenção

Quando queremos acessar um campo de um registro e fazemos isso por meio de um ponteiro passado por referência, devemos colocá-lo entre parênteses.

Por exemplo: (*inicio)->info

Sem os parênteses, o compilador poderia achar que quiséssemos avaliar *(inicio->info), e não é isso o que desejamos!



Vamos simular a execução da função acompanhando o que ocorre na memória. Inicialmente, vamos supor que a lista ainda esteja vazia e, portanto, o valor da variável *inicio*, alocada no endereço 100, seja NULL, como mostra a [Figura 7.1a](#).

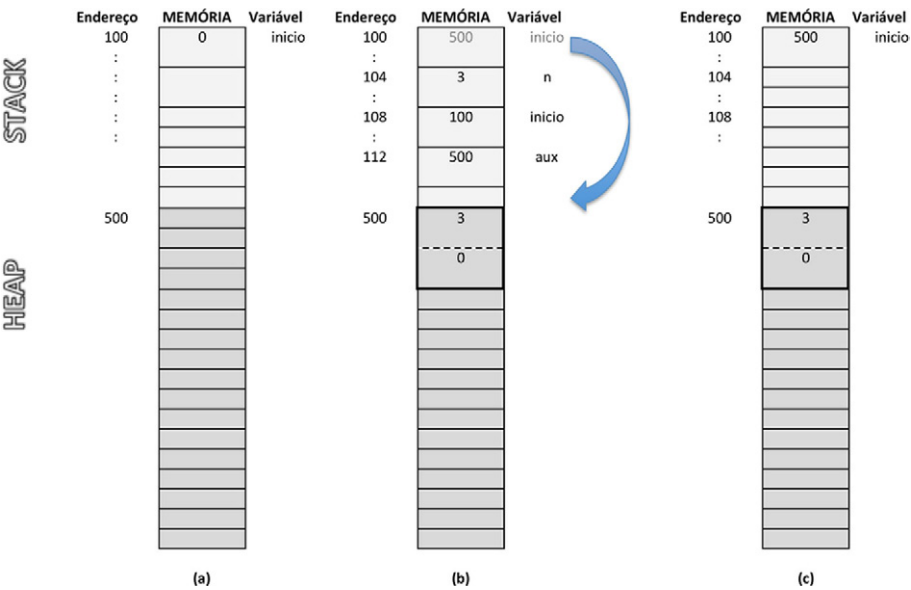


FIGURA 7.1: Inserção do primeiro nó na lista.

A função `insere_inicio`¹ é chamada com dois parâmetros: um inteiro, que é o valor a ser inserido, e o ponteiro para o primeiro nó, passado por referência, isto é, o endereço do ponteiro:

```
insere_inicio (3, &inicio);
```

Na execução da função, seus parâmetros são alocados na pilha. O primeiro parâmetro, *n*, é alocado no endereço 104 de memória e recebe o valor 3 (passado na chamada da função). O segundo parâmetro, *inicio*, é alocado no endereço 108 e recebe o valor 100 (a função principal passa o endereço de sua variável *inicio*, que está alocada no endereço 100). Esses parâmetros são mostrados na [Figura 7.1b](#).

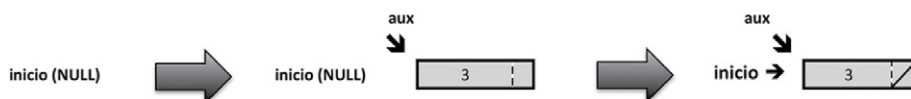
Atenção



Na execução da função `insere_inicio`, sempre que nos referirmos à variável *inicio* estaremos nos referindo à cópia (alocada no endereço 108). A variável *inicio* da *main* não é acessada diretamente. Para não se confundir, muitos preferem dar nomes diferentes aos parâmetros.

O primeiro comando executado pela função é o `malloc`, que vai alocar 8 bytes a partir do endereço 500. Esse valor é armazenado na variável *aux*. Como *aux* não é `NULL`, seu campo *info* será preenchido com o valor 3 (lembre-se de que *aux* vale 500 e a seta faz com que seja seguido um ponteiro até esse endereço e lá seja feita a alteração).

Depois disso, a função pega o valor de *inicio* (100) e, em seguida, seu conteúdo - * (o conteúdo do endereço 100 é zero). Como `not inicio(!inicio)` é verdadeiro, ou seja, a lista está vazia, a função faz `*inicio` (o conteúdo do endereço 100) ser igual a *aux*, que vale 500. Em seguida, faz o campo *prox* do registro alocado no endereço 500 ser igual a `NULL`. Terminada a execução da função, os parâmetros e as variáveis locais da função são desalocados, ou seja, liberados da memória. A [Figura 7.1c](#) mostra o estado final da memória. A seguir, veremos também um esquema que mostra o estado inicial da lista, o novo nó alocado (*aux*) e o estado final da lista após a inserção.



1. Tecnicamente, `void` é um tipo de dados da linguagem C. Como `insere_inicio` retorna um `void`, vou chamá-la de *função*, mas, se você preferir, pode chamar de *procedimento*.

Dica

Para que uma função possa alterar o valor de uma variável da *main*, passamos como parâmetro por referência, ou seja, o endereço do parâmetro, identificado pelo símbolo &.

Dentro da função, simplesmente usamos os parâmetros passados por referência antecidos pelo símbolo * (que representa o conteúdo, acessando assim o conteúdo da variável da *main*).

Quando passamos os parâmetros por valor, acessamos as cópias das variáveis. As alterações que fizermos nas cópias serão perdidas quando essas cópias forem desalocadas no final da execução da função.



Vamos inserir mais um elemento na lista através da seguinte chamada:

```
insere_inicio (2, &inicio);
```

Nossa lista é inicialmente mostrada na [Figura 7.2a](#). Com a chamada, são feitas as cópias dos parâmetros *n* e *inicio*, e é alocada a variável local *aux* na pilha.

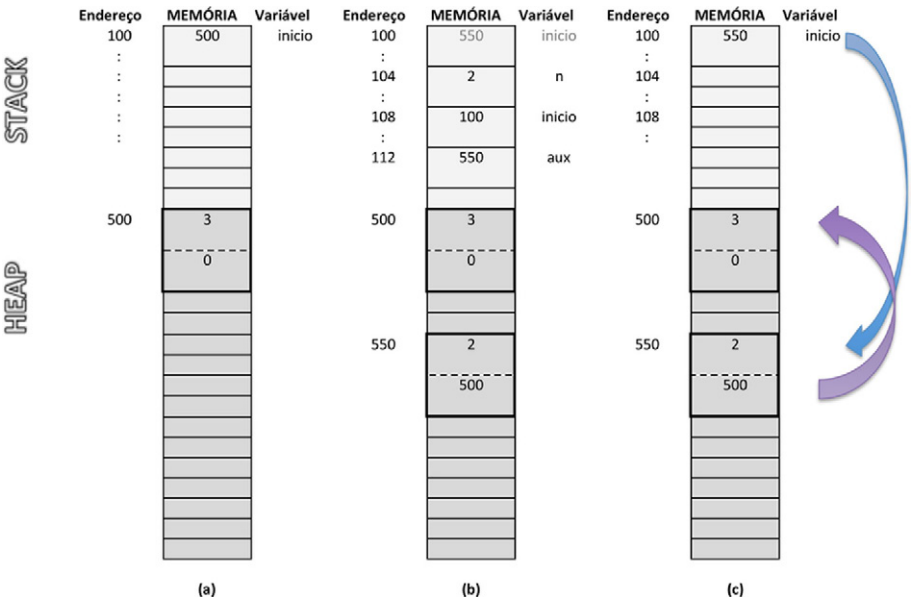
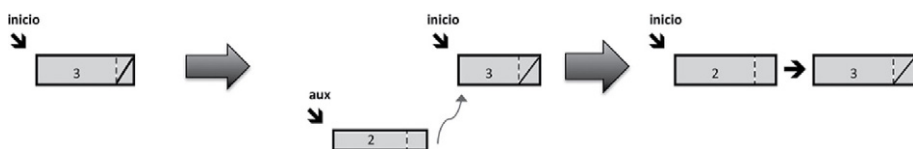


FIGURA 7.2: Inserção do segundo nó na lista.

Acompanhe a execução da função *insere_inicio*: a execução do comando *malloc* aloca 8 bytes a partir do endereço 550, que é atribuído à variável *aux*. Como o comando *malloc* não retornou NULL, o campo *info* presente no endereço 550 será preenchido com o valor 2.

Em seguida, a função testa o valor do conteúdo de *inicio*. Agora, o conteúdo do endereço 100 não é zero (é 500), ou seja, a lista não está mais vazia. Assim, ele insere o valor 500 no campo *prox* do registro apontado por *aux*, ou seja, o registro que está no endereço 550. Por fim, a função insere o valor de *aux* (550) como conteúdo do endereço 100. O resultado pode ser visto na Figura 7.2b.

A Figura 7.2c mostra a memória após a execução da função e a liberação da memória ocupada pelos parâmetros e pela variável local. A seguir, veremos também como ocorreu a inserção: o estado inicial da lista, a alocação do novo nó (*aux*) e as ligações realizadas com os ponteiros.



Inserção de um nó no final da lista

Para inserir um novo nó de valor *n* no final da lista ligada, vamos usar o mesmo raciocínio: no caso mais simples (quando a lista estiver vazia), a nova lista terá apenas o novo nó. Assim, a implementação do caso mais simples é igual à inserção de um nó no início de uma lista vazia. No caso geral (lista não vazia), vamos inserir o novo elemento após o último nó da lista. Para encontrá-lo precisaremos percorrer a lista em busca do último nó. Mas como poderemos reconhecer o último elemento? Bem, vamos observar uma lista para ver o que o último elemento tem de diferente em relação aos outros nós:



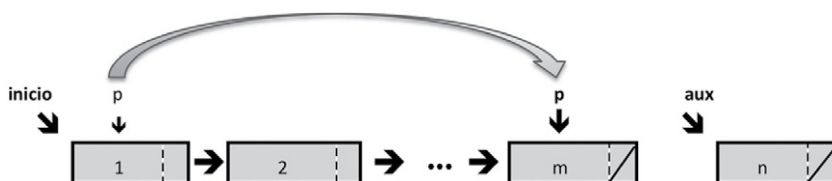
Como você percebeu, o campo *prox* do último elemento tem o valor NULL. Como não sabemos quantos elementos a lista tem, vamos procurar esse nó através de um comando *while* para, então, inserir o novo nó. A função fica assim:

```

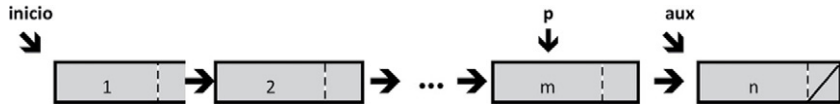
void insere_fim (int n, no **inicio)
{
no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
if (aux)                               // conseguiu alocar espaço
{
    aux->info = n;
    aux->prox = NULL; // como o novo nó será inserido no final da lista, ele não terá ninguém à frente
    if (!(*inicio)) // lista vazia
        (*inicio) = aux;
    else // lista não vazia
    {
        no *p = (*inicio); // usaremos o ponteiro p para encontrar o último nó
        while (p->prox != NULL)
            p = p->prox;
        p->prox = aux; // ligaremos o último nó ao novo nó da lista
    }
}
else printf ("Heap overflow\n");
}

```

No caso geral, após ter alocado espaço e preenchido o novo nó (*aux*), a função inicializa o ponteiro *p* no primeiro nó (*inicio*) e, depois, executa o *loop*. O comando *while* faz o ponteiro *p* armazenar os endereços de todos os nós da lista, um a um. Inicialmente, *p* é inicializado com o valor do primeiro nó (*inicio*). Se esse for o único elemento da lista, seu campo *prox* será *NULL*, e o comando *while* não fará nada. Se houver mais nós, o valor do ponteiro *p* será trocado pelo campo *prox* do primeiro nó. Nesse campo, *prox* está armazenando o endereço do segundo nó. Dessa forma, o ponteiro *p* percorrerá toda a lista até chegar ao último elemento, cujo campo *prox* é *NULL*. A ilustração a seguir mostra a execução desse trecho do código.



Agora é fácil! Para ligar a lista ao novo nó, basta inserir no campo *prox* do último elemento o endereço do novo nó (*aux*), que está no final da função. O resultado pode ser visto a seguir:



Dica

Se precisarmos inserir muitos elementos no final de uma lista muito grande, poderemos ter que percorrê-la muitas vezes, o que pode levar um tempo considerável. Nesse caso, podemos trabalhar com dois ponteiros por lista: um para armazenar o endereço do primeiro elemento e outro para armazenar o endereço do último.

Mas lembre-se: é preciso manter o segundo ponteiro sempre atualizado, e, para isso, pode ser necessário passá-lo como parâmetro para as funções, assim como passamos o ponteiro de início da lista!

Impressão de todos os nós da lista

Agora que já vimos como percorrer uma lista ligada, vamos praticar. Primeiro, vamos percorrer uma lista imprimindo todos os valores armazenados.

```
void imprime (no *inicio)
{
    while (inicio)    // enquanto houver lista
    {
        printf("%d ", inicio->info);    // imprime o valor do nó
        inicio=inicio->prox;            // anda para o próximo nó
    }
}
```

Vamos analisar os detalhes da função. Primeiro, ela percorre a lista enquanto *inicio*, ou seja, enquanto o ponteiro *inicio* for *TRUE*, isto é, diferente de *NULL* (zero). Nessa função não usamos um ponteiro auxiliar para percorrer a lista, mas o próprio ponteiro *inicio*. Mas ao mudar o ponteiro *inicio* não perderemos os primeiros elementos? A resposta é NÃO, porque vamos alterar a cópia do ponteiro, realizada quando a função foi chamada (lembre-se de que, ao se executar uma função, são criadas instâncias novas para seus parâmetros e variáveis locais, que são desalocadas no final da execução da função).

Caso a lista esteja inicialmente vazia, o ponteiro *inicio* será *NULL*, e os comandos dentro do *while* não serão executados. Por isso, não testamos explicitamente o caso mais simples (lista vazia) – que está no código porém oculto – e partimos diretamente para o caso geral (lista não vazia).

Dentro do comando *while*, o campo *info* do ponteiro *inicio* será impresso, e, em seguida, *inicio* passa para o próximo nó.

Um erro bastante comum na implementação dessa função é testar *while* (*inicio->prox!=NULL*). Testando *inicio->prox* em vez de *inicio* fará com que a função não imprima o último elemento, pois nele o campo *inicio->prox* já será *NULL*. E o pior ocorrerá se executarmos a função com uma lista inicialmente vazia. Quando a função testar *inicio->prox*, ela tentará avaliar o resultado de *NULL->...*, que dá um erro fatal (o programa “aborta” a execução).

Encontrar um elemento da lista

Outra função muito importante no uso de listas ligadas é verificar se um elemento *n* está presente na lista. Claro que, se a lista estiver vazia, o elemento procurado não estará presente. No caso de encontrarmos o elemento, já podemos parar a execução da função, pois continuar percorrendo a lista só nos faria perder tempo. Vamos ver uma das possíveis implementações para essa função:

```
int pertence (int n, no *inicio)
{
    while (inicio)
    {
        if (inicio->info == n) return (1); //encontrou o elemento
        inicio=inicio->prox;
    }
    return (0); // acabou a lista sem encontrar o elemento
}
```

Essa versão só nos responde com 1 ou 0 (*TRUE* ou *FALSE*) se o elemento está ou não na lista. Em alguns compiladores poderíamos usar o tipo *bool* (*booleano*) para essa função, no lugar do tipo *int*.

No entanto, na maioria das aplicações, os nós armazenarão mais dados sobre o registro (além do código, poderíamos armazenar atributos como descrição, fabricante, preço etc.). Nesse caso, a função precisa dar acesso a esses dados, e uma forma prática de fazer isso é por meio do ponteiro para o nó. A versão a seguir retorna o endereço para o nó cuja chave de busca é *n*. A função que a chama (*main*, por exemplo) deve testar se o valor retornado

é NULL antes de tentar o acesso aos dados. O programa apresentado no final deste capítulo mostra como utilizar o ponteiro retornado.

```
no* busca (int n, no *inicio)
{
    while (inicio)
    {
        if (inicio->info == n) return (inicio); //encontrou o elemento
        inicio=inicio->prox;
    }
    return (NULL); // acabou a lista sem encontrar o elemento.
}
```

Remoção do primeiro nó da lista

Para manipular nossa primeira estrutura de dados dinâmica, precisamos excluir alguns de seus elementos. Podemos fazer isso no início, no meio ou no final da lista. Nos exemplos a seguir, vamos retornar o valor do nó removido, mas, se precisássemos usar os outros campos, retornaríamos o ponteiro para o nó, como no exemplo anterior.

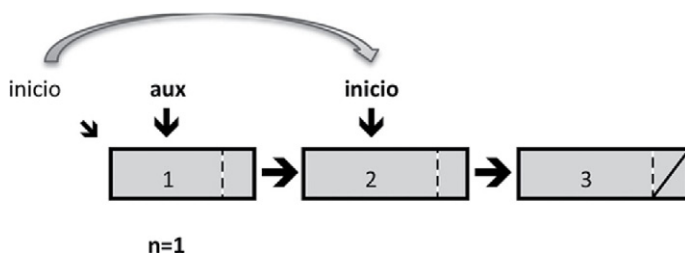
Na remoção do primeiro elemento da lista, consideraremos o caso de a lista estar vazia (caso mais simples), em que não teremos de fazer nada, apenas retornar uma *flag* para informar que não havia elementos na lista. No caso geral, removeremos o primeiro elemento e liberaremos a memória ocupada por ele. Vamos também atualizar a lista (o segundo nó passará a ser o primeiro; o terceiro passará a ser o segundo, e assim por diante).

```
int remove_inicio (no **inicio)
{
    if (!(*inicio)) return (-1); // retorna a flag -1 para informar que a lista estava vazia
    no *aux=(*inicio);           // guarda o primeiro nó em aux
    int n=(*inicio)->info;       // guarda o valor do nó para retornar
    (*inicio)=(*inicio)->prox; // anda com inicio para o segundo nó
    free (aux);                  // libera o espaço de memória ocupado pelo nó removido
    return(n);                   // retorna o valor do nó removido
}
```

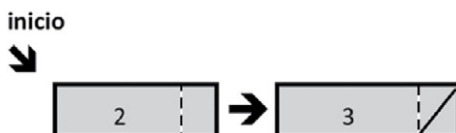
Vamos observar o funcionamento nas ilustrações. Suponhamos que a lista inicial seja a seguinte:



1. Como a lista não está vazia, a função inicializa as variáveis *aux* e *n* para guardar o nó inicial e seu valor. Em seguida, anda com o ponteiro *inicio*:



2. Depois disso, a função libera o espaço de memória do ponteiro *aux* antes de retornar. A lista fica da seguinte forma:



3. Nesse caso, a lista foi alterada, pois o ponteiro *inicio* foi passado por referência, ao contrário dos algoritmos de percurso. Além disso, a função separa o caso mais simples (lista vazia) do caso geral (lista não vazia), mas não usa *else* entre eles, porque, no caso de a lista ser vazia, a função já retorna à principal e, portanto, não executa os comandos abaixo do *return*. Assim, o uso do *else* é opcional.

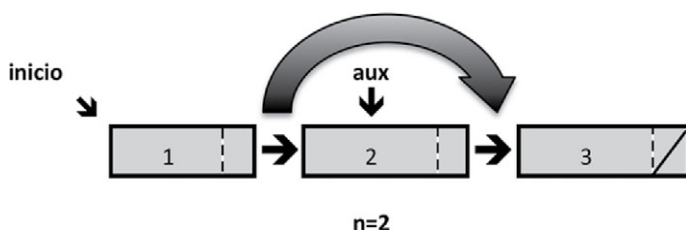
Remoção do segundo nó da lista

Para remover um nó intermediário, devemos tomar cuidado para manter os ponteiros ligando corretamente todos os nós após a remoção. Para remover o segundo nó precisamos ligar o campo *prox* do primeiro nó ao terceiro. Assim, nas remoções de nós intermediários, precisamos guardar o nó anterior ao removido em um ponteiro.

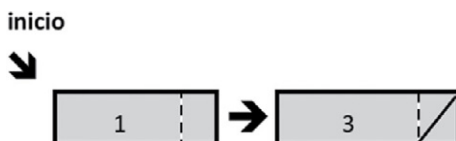
```
int remove_segundo (no **inicio)
{
    if (!(*inicio) || !(*inicio)->prox) return (-1); // a lista tinha menos de dois elementos
    no *aux=(*inicio)->prox; // guarda o nó a ser removido em aux
    int n=(aux)->info; // guarda o valor do nó para retornar
    (*inicio)->prox=(aux)->prox; // liga o nó anterior ao removido ao nó posterior
    free(aux); // libera o espaço de memória ocupado pelo nó removido
    return(n); // retorna o valor do nó removido
}
```



Nessa função, a lista não terá segundo nó no caso de estar vazia ou conter apenas um elemento. Nesses casos, a função retornará a *flag* -1. Caso contrário, guardará o nó a ser removido e seu valor nas variáveis *aux* e *n*, respectivamente, e depois mudará o valor do campo *prox* de *inicio*. Observe a ilustração a seguir:



Após liberar a memória ocupada pelo registro apontado por *aux* e retornar, a lista resultante é a seguinte:



Percebemos que essa função não altera o valor do ponteiro *inicio*. Assim, ele também pode ser passado por valor.

Remoção de um nó do final da lista

Para remover um nó do final da lista, temos de considerar alguns casos: a lista está vazia e, assim, não há nada a remover; a lista tem apenas um nó e, com sua remoção, se tornará vazia; a lista tem dois ou mais elementos.

Por que precisamos considerar o caso de a lista ter um único elemento? Como sabemos, para remover um nó, precisamos guardar o endereço do nó anterior ao nó a ser removido (para ligá-lo ao nó posterior). Quando a lista tem apenas um nó, o nó removido não tem anterior, e, assim, vamos tratar esse caso separadamente.

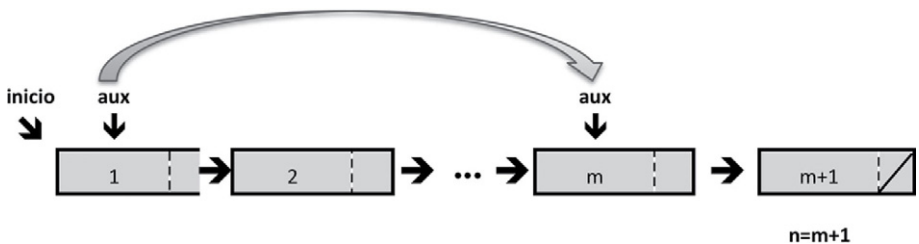
```
int remove_ultimo (no **inicio)
{
    int n;
    if (!(*inicio)) return (-1);    // lista vazia
    if (!(*inicio)->prox)          // lista com apenas um nó
    {
        n=(*inicio)->info;
        free(*inicio);
        (*inicio)=NULL;           // após a liberação do único nó, a lista passa a ser vazia
        return(n);
    }
    no *aux=(*inicio);             // inicializa aux no início da lista
    while (aux->prox->prox)         // anda com aux até o penúltimo nó
        aux=aux->prox;
    n=aux->prox->info;              // guarda o valor do último nó
    free(aux->prox);               // libera a memória do último nó
    aux->prox=NULL;               // atualiza o ponteiro do penúltimo nó, que passou a ser o último
    return(n);                   // retorna o valor do nó removido
}
```

No caso da remoção do único nó da lista, ela se tornará vazia, portanto devemos assegurar que seu valor seja NULL.

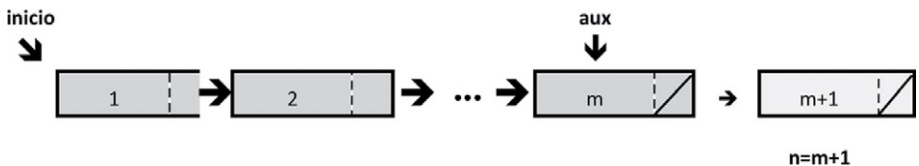
Para ilustrar o funcionamento do caso geral, vamos considerar a seguinte lista com $m+1$ elementos:



A função inicializa o ponteiro *aux* no nó inicial e anda até o penúltimo nó (até que *aux->prox->prox* seja NULL). Ela, então, armazena o valor do último nó na variável *n*:



A seguir, o nó *aux->prox* é liberado, e o ponteiro *prox* de *aux* passa a valer NULL. Essas etapas devem ser realizadas nessa ordem, para não se perder o endereço do último nó antes de liberar a memória). Por fim, a função retorna o valor do nó removido.



Dica

Para permitir a recuperação futura de um registro, podemos removê-lo da lista e inseri-lo em uma lista de elementos removidos, que manterá os dados do registro original, facilitando a recuperação.

É possível também usar uma *flag* em cada registro, identificando-o como *ativo* ou *removido*. Assim, quando um registro for removido da lista, apenas alteramos o valor da *flag* (de 1 para 0, por exemplo) e o mantemos na lista original. Assim, fazemos apenas uma *remoção lógica*. Quando formos imprimir os elementos da lista, imprimiremos apenas os nós cuja *flag* indicar que o elemento está ativo.



Para terminar essa estrutura de dados, vamos ver um exemplo de programa de manipulação de listas ligadas. O Código 7.1 reúne as funções desenvolvidas neste capítulo, que são acessadas por meio de um menu de opções. Os dados armazenados são inseridos no arquivo *lista.txt* sempre que o programa é encerrado. Na inicialização do programa, esse arquivo é carregado.

Código 7.1

```
#include <stdio.h>
#include <stdlib.h>

typedef struct tipo_no no;      // DEFINICAO DA ESTRUTURA
struct tipo_no {
    int info;
    struct tipo_no *prox;
};

int display_menu(void);        // PROTOTIPOS DAS FUNCOES
void insere_inicio (int , no **);
void insere_fim (int, no **);
void imprime (no *);
int pertence (int, no *);
no* busca (int, no *);
int remove_inicio (no **);
int remove_segundo (no **);
int remove_ultimo (no **);

int main()                    // PROGRAMA PRINCIPAL
{
    FILE *arq;
    int opcao, num;
    no *lista = NULL; // inicialização da lista
    no *aux;
    no data;

    if((arq=fopen("lista.txt","r+")) == NULL) // tenta abrir o arquivo para leitura
    {
        printf("Arquivo nao existente, sera criado\n");
        arq=fopen("lista.txt","w+");          // cria o arquivo caso nao existir
    }
    else {
        printf("Leitura do arquivo....\n");
        while (!feof(arq))                    // le ate o final do arquivo
        {
            fread(&data ,sizeof(no), 1, arq); // le um no
            if (!feof(arq))                    // se nao tiver chegado no final do arquivo
            {
                printf("Insercao do registro %d\n",data.info);
                insere_fim(data.info, &lista);
            }
        }
    }
}
```

(Continua)

```

do { // CHAMADA DA FUNCAO ESCOLHIDA PELO USUARIO
    opcao=display_menu();
    switch (opcao) {
        case 1:
            printf("\nValor do no:");
            scanf("%d",&num);
            insere_inicio(num, &lista);
            break;
        case 2:
            printf("\nValor do no:");
            scanf("%d",&num);
            insere_fim(num, &lista);
            break;
        case 3:
            imprime(lista);
            break;
        case 4:
            printf("\nValor que deseja procurar:");
            scanf("%d",&num);
            if (pertence(num, lista)) printf("Elemento pertence a lista\n");
            else printf("Elemento nao pertence a lista\n");
            break;
        case 5:
            printf("\nValor que deseja procurar:");
            scanf("%d",&num);
            aux=busca(num, lista);
            if (aux) printf("Acesso ao valor do elemento: %d\n", aux->info);
            else printf("Elemento nao esta na lista\n");
            break;
        case 6:
            num=remove_inicio(&lista);
            if (num==-1) printf("Lista vazia\n");
            else printf("Elemento removido: %d\n", num);
            break;
        case 7:
            num=remove_segundo(&lista);
            if (num==-1) printf("Nao ha dois elementos na lista\n");
            else printf("Elemento removido: %d\n", num);
            break;
        case 8:
            num=remove_ultimo(&lista);
            if (num==-1) printf("Lista vazia\n");
            else printf("Elemento removido: %d\n", num);
            break;
    }
} while (opcao != 0);

rewind(arq); // volta o ponteiro de arquivo ate seu inicio
fflush(arq); // limpa o arquivo
aux=lista;

```

(Continua)


```

while (aux!=NULL){ //percorre a lista ate o final
    printf("Gravando o registro %d\n",aux->info);
    fwrite((char *)aux,sizeof(no),1,arq); // grava o no no arquivo
    aux = aux -> prox;
}

printf("Fim");
fflush(stdin);
scanf("%c",&opcao); // somente para acompanhar a execucao do programa
fclose(arq); // fecha o arquivo
}

// IMPRIME AS OPCOES DO MENU
int display_menu(void)
{
    int opcao;
    printf("\n\n ----- \n\n");
    printf("\n LISTA SIMPLES \n\n OPCOES: \n\n");
    printf("1- Inserir no inicio\n");
    printf("2- Inserir no final\n");
    printf("3- Imprimir toda a lista\n");
    printf("4- Buscar um elemento - booleano\n");
    printf("5- Buscar um elemento - ponteiro\n");
    printf("6- Remover o primeiro elemento\n");
    printf("7- Remover o segundo elemento\n");
    printf("6- Remover o ultimo elemento\n");
    printf("0- Sair (grava em arquivo) \n\n => ");
    scanf("%d",&opcao);
    return(opcao);
}

// INSERE UM NO DE VALOR N NO INICIO DA LISTA
void insere_inicio (int n, no **inicio)
{
    no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
    if (aux){ // conseguiu alocar espaço
        aux->info = n;
        if (!(*inicio)) { // lista vazia - é o mesmo que if(*inicio==NULL)
            (*inicio) = aux;
            (*inicio)->prox = NULL;
        }
        else { // lista não vazia
            aux->prox = (*inicio);
            (*inicio) = aux;
        }
    }
    else printf ("Heap overflow\n");
}

```

(Continua)

```

return (NULL); // acabou a lista sem encontrar o elemento
}

// REMOVE O PRIMEIRO ELEMENTO DA LISTA
int remove_inicio (no **inicio)
{
    if (!(*inicio)) return (-1); // retorna a flag -1 para informar que a lista estava vazia
    no *aux=(*inicio); // guarda o primeiro nó em aux
    int n=(*inicio)->info; // guarda o valor do nó para retornar
    (*inicio)=(*inicio)->prox; // anda com inicio para o segundo nó
    free(aux); // libera o espaço de memória ocupado pelo nó removido
    return(n); // retorna o valor do nó removido
}

// REMOVE O SEGUNDO ELEMENTO DA LISTA
int remove_segundo (no **inicio)
{
    if (!(*inicio)|| !(*inicio)->prox) return (-1); // a lista tinha menos de dois elementos
    no *aux=(*inicio)->prox; // guarda o nó a ser removido em aux
    int n=(aux)->info; // guarda o valor do nó para retornar
    (*inicio)->prox=(aux)->prox; // liga o nó anterior ao removido ao nó posterior
    free(aux); // libera o espaço de memória ocupado pelo nó removido
    return(n); // retorna o valor do nó removido
}

// REMOVE O ÚLTIMO ELEMENTO DA LISTA
int remove_ultimo (no **inicio)
{
    int n;
    if (!(*inicio)) return (-1); // lista vazia
    if (!(*inicio)->prox) // lista com apenas um nó
    {
        n=(*inicio)->info;
        free(*inicio);
        (*inicio)=NULL;
        return(n);
    }
    no *aux=(*inicio); // inicializa aux no início da lista
    while (aux->prox->prox) // anda com aux até o penúltimo nó
        aux=aux->prox;
    n=aux->prox->info; // guarda o valor do último nó
    free(aux->prox); // libera a memória do último nó
    aux->prox=NULL; // retorna o valor do nó removido
    return(n);
}

```

(Continua)

```

return (NULL); // acabou a lista sem encontrar o elemento
}

// REMOVE O PRIMEIRO ELEMENTO DA LISTA
int remove_inicio (no **inicio)
{
    if (!(*inicio)) return (-1); // retorna a flag -1 para informar que a lista estava vazia
    no *aux=(*inicio); // guarda o primeiro nó em aux
    int n=(*inicio)->info; // guarda o valor do nó para retornar
    (*inicio)=(*inicio)->prox; // anda com inicio para o segundo nó
    free(aux); // libera o espaço de memória ocupado pelo nó removido
    return(n); // retorna o valor do nó removido
}

// REMOVE O SEGUNDO ELEMENTO DA LISTA
int remove_segundo (no **inicio)
{
    if (!(*inicio)|| !(*inicio)->prox) return (-1); // a lista tinha menos de dois elemento
    no *aux=(*inicio)->prox; // guarda o nó a ser removido em aux
    int n=(aux)->info; // guarda o valor do nó para retornar
    (*inicio)->prox=(aux)->prox; // liga o nó anterior ao removido ao nó posterior
    free(aux); // libera o espaço de memória ocupado pelo nó removido
    return(n); // retorna o valor do nó removido
}

// REMOVE O ÚLTIMO ELEMENTO DA LISTA
int remove_ultimo (no **inicio)
{
    int n;
    if (!(*inicio)) return (-1); // lista vazia
    if (!(*inicio)->prox) // lista com apenas um nó
    {
        n=(*inicio)->info;
        free(*inicio);
        (*inicio)=NULL;
        return(n);
    }
    no *aux=(*inicio); // inicializa aux no início da lista
    while (aux->prox->prox) // anda com aux até o penúltimo nó
        aux=aux->prox;
    n=aux->prox->info; // guarda o valor do último nó
    free(aux->prox); // libera a memória do último nó
    aux->prox=NULL; // libera a memória do último nó
    return(n); // retorna o valor do nó removido
}

```



Vamos programar

Java

Código 7.1

```
public static void main(String[] args) throws FileNotFoundException, IOException {

    LinkedList list = new LinkedList(); // Instanciar a lista ligada

    File arquivo = new File("c:/java/lista_ligada.txt");
    FileReader fr = new FileReader(arquivo);
    BufferedReader br = new BufferedReader(fr);
    while (br.ready()) {
        list.add(br.readLine());
    }
    br.close();
    fr.close();

    do {
        display_menu();
        Scanner entrada = new Scanner(System.in);
        opcao = Integer.parseInt(entrada.nextLine());
        switch (opcao) {
            case 1:
                System.out.print("Valor do no: ");
                list.addFirst(Integer.parseInt(entrada.nextLine()));
                break;
            case 2:
                System.out.print("Valor do no: ");
                list.addLast(Integer.parseInt(entrada.nextLine()));
                break;
            case 3:
                System.out.println("LinkedList: " + list); // Imprimir a lista (função imprime)
                break;
            case 4:
                System.out.print("Valor que deseja procurar: ");
                boolean pertence = list.contains(Integer.parseInt(entrada.nextLine()));
                if (pertence) {
                    System.out.println("Elemento pertence a lista");
                } else {
                    System.out.println("Elemento nao pertence a lista");
                }
                break;
        }
    } while (true);
}
```

(Continua)

```
public static void main(String[] args) throws FileNotFoundException, IOException {

    LinkedList list = new LinkedList(); // Instanciar a lista ligada

    File arquivo = new File("c:/java/lista_ligada.txt");
    FileReader fr = new FileReader(arquivo);
    BufferedReader br = new BufferedReader(fr);
    while (br.ready()) {
        list.add(br.readLine());
    }
    br.close();
    fr.close();

    do {
        display_menu();
        Scanner entrada = new Scanner(System.in);
        opcao = Integer.parseInt(entrada.nextLine());
        switch (opcao) {
            case 1:
                System.out.print("Valor do no: ");
                list.addFirst(Integer.parseInt(entrada.nextLine()));
                break;
            case 2:
                System.out.print("Valor do no: ");
                list.addLast(Integer.parseInt(entrada.nextLine()));
                break;
            case 3:
                System.out.println("LinkedList: " + list); // Imprimir a lista (função imprime)
                break;
            case 4:
                System.out.print("Valor que deseja procurar: ");
                boolean pertence = list.contains(Integer.parseInt(entrada.nextLine()));
                if (pertence) {
                    System.out.println("Elemento pertence a lista");
                } else {
                    System.out.println("Elemento nao pertence a lista");
                }
                break;
        }
    }
}
```

(Continua)

```

        break;
    }
} while (opcao != 0);
}

public static void display_menu() {
    System.out.println();
    System.out.println();
    System.out.println(" ----- ");
    System.out.println("1- Inserir no inicio");
    System.out.println("2- Inserir no final");
    System.out.println("3- Imprimir toda a lista");
    System.out.println("4- Buscar um elemento - booleano");
    System.out.println("5- Buscar um elemento - ponteiro");
    System.out.println("6- Remover o primeiro elemento");
    System.out.println("7- Remover o segundo elemento");
    System.out.println("8- Remover o ultimo elemento");
    System.out.println("0- Sair (grava em arquivo)");
}

```

Python

Código 7.1

```

#DEFINICAO DA ESTRUTURA
class No :
    def __init__(self, info=None, prox=None):
        self.info = info
        self.prox = prox
    #toString()
    def __str__(self):
        return str('Info:%s\n' %(self.info))

class Lista:
    def __init__(self):
        self.inicio = None

#INSERE UM NO DE VALOR N NO INICIO DA LISTA
def insere_inicio(self, num):
    aux = No() #aloca espaço para o novo nó
    aux.info = num
    if self.inicio == None:
        self.inicio = aux
        self.inicio.prox = None
    else: #lista não vazia
        aux.prox = self.inicio
        self.inicio = aux

```

(Continua)


```
#INSERE UM NO DE VALOR N NO FINAL DA LISTA
def insere_fim(self, n):
    aux = No() #aloca espaço para o novo no
    aux.info = n
    if self.inicio == None:
        self.inicio = aux
    else:
        p = self.inicio #usaremos o ponteiro p para encontrar o ultimo no
        while(p.prox):
            p=p.prox #anda para o proximo no
        p.prox=aux

#VERIFICA SE EXISTE UM NO DE VALOR N NA LISTA - RETORNA BOOLEANO
def pertence(self, n):
    aux=self.inicio
    while(aux): #enquanto houver lista
        if aux.info == n:
            return 1
        aux=aux.prox #anda para o proximo no
    return 0

#VERIFICA SE EXISTE UM NO DE VALOR N NA LISTA - RETORNA PONTEIRO
def busca(self, n):
    aux=self.inicio
    while aux:
        if aux.info == n: #encontrou o elemento
            return hex(id(aux))
        aux=aux.prox
    return None #acabou a lista sem encontrar o elemento

#IMPRIME TODOS OS ELEMENTOS DA LISTA
def imprime(self):
    print "Elementos da lista: "
    aux = self.inicio
    while(aux):
        print aux.info
        aux=aux.prox
```

(Continua)

```

#REMOVE O PRIMEIRO ELEMENTO DA LISTA
def remove_inicio(self):
    if not self.inicio:
        return -1; #retorna a flag -1 para informar que a lista estava vazia
    #aux = self.inicio
    n = self.inicio.info #guarda o valor do no para retornar
    self.inicio=self.inicio.prox #anda com inicio para o segundo no
    return n #retorna o valor do no removido

#REMOVE O SEGUNDO ELEMENTO DA LISTA
def remove_segundo(self):
    if not self.inicio or not self.inicio.prox: #a lista tinha menos de dois elemento
        return -1
    aux = inicio.prox #guarda o no a ser removido em aux
    n=aux.info #guarda o valor do no para retornar
    inicio.prox=aux.prox #liga o no anterior ao removido ao no posterior
    return n #retorna o valor do no removido

#REMOVE O ULTIMO ELEMENTO DA LISTA
def remove_ultimo(self):
    if not self.inicio: #lista vazia
        return -1
    if not self.inicio.prox: #lista com apenas um no
        n=self.inicio.info
        self.inicio=None
        return n
    aux=self.inicio
    while aux.prox.prox: #anda com aux ate o penultimo no
        aux=aux.prox
    n=aux.prox.info #guarda o valor do ultimo no
    aux.prox=None #libera a memoria do ultimo no
    return n #retorna o valor do no removido

def display_menu():
    print("\n\n -----")
    print("\nLISTA SIMPLES \n\nOPCOES: ")
    print("1- Inserir no inicio")
  
```

(Continua)


```

print("2- Inserir no final")
print("3- Imprimir toda a lista")
print("4- Buscar um elemento - booleano")
print("5- Buscar um elemento - ponteiro")
print("6- Remover o primeiro elemento")
print("7- Remover o segundo elemento")
print("8- Remover o ultimo elemento")
print("0- Sair (grava em arquivo) => \n")
entrada = int(input('Opcao: '))
return entrada

lista = Lista() #inicializacao da lista
try:
    f=open( 'arq.txt', "r+") #tenta abrir o arquivo para leitura
except:
    f=open( 'arq.txt', "w+") #cria o arquivo caso nao existir
for linha in f.readlines(): #leia o final do arquivo
    lista.inserir_fim(int(linha)) #leu um no
f.close()
opcao=-1
while opcao <> 0 : #CHAMADA DA FUNCAO ESCOLHIDA PELO USUARIO
    opcao=display_menu();
    if opcao == 1 :
        num = int(input('Numero: '))
        lista.inserir_inicio(num)
    elif opcao == 2:
        num = int(input('Numero: '))
        lista.inserir_fim(num)
    elif opcao == 3:
        lista.imprime()
    elif opcao == 4:
        num = int(input('Numero: '))
        print lista.pertence(num)
    elif opcao == 5:
        num = int(input('Numero: '))
        print lista.busca(num)

```

(Continua)

```
elif opcao == 6:
    print lista.remove_inicio()
elif opcao == 7:
    print lista.remove_segundo()
elif opcao == 8:
    print lista.remove_ultimo()
f = open( 'arq.txt', 'w+')
aux=lista.inicio
while aux:
    #f.write(str(aux.info))
    print >> f, str(aux.info) #gravacao no arquivo
    aux=aux.prox
f.close() #fechao arquivo
```



Para fixar

Implemente as seguintes funções para listas ligadas:

- função que retorna a quantidade de nós de uma lista ligada;
- função que insere um elemento de valor n na i -ésima posição da lista (por exemplo, insere o valor 15 na 3ª posição da lista);
- função que remove o i -ésimo elemento da lista ligada.



Para saber mais

Você pode ter outros detalhes sobre listas ligadas lendo o Capítulo 4 do livro *Algoritmos e estruturas de dados* (Wirth, 1999). Nele você encontrará ótimos exemplos de algoritmos na linguagem Pascal.



Navegar é preciso

Você pode encontrar bastante material sobre listas ligadas na internet, por exemplo, acessando o link

<http://pt.kioskea.net/faq/10263-listas-simplesmente-encadeadas>. Nele há outros algoritmos para manipulação de listas ligadas simples.

Exercícios

1. Muitas vezes, queremos armazenar os registros em ordem crescente de códigos ou em ordem alfabética, por exemplo. Implemente uma função que insira um registro de valor n em uma lista ligada ordenada. Observe que, se partirmos de uma lista vazia e só usarmos essa função para inserir elementos, a lista estará sempre ordenada – que é o princípio da ordenação por inserção.
2. Muitas aplicações devem permitir a remoção de registros baseada em seu valor (sua chave de busca). Implemente uma função que remova um elemento de valor n da lista ligada.

Glossário

Modularidade: propriedade dos sistemas desenvolvidos em módulos (funções, subsistemas,...) independentes. Isso facilita a detecção e a correção de erros de implementação, e também a substituição de partes (módulos), sem afetar o funcionamento do restante do sistema.

Referências bibliográficas

WIRTH, N. *Algoritmos e estruturas de dados*. São Paulo: LTC, 1999.



QUE VEM DEPOIS

As listas ligadas lineares oferecem vantagens como a alocação dinâmica de memória e a facilidade de inserção de elementos no meio da lista, mas trazem algumas dificuldades, como em casos da remoção de elementos, em que precisamos guardar o nó anterior ao removido. Existem outros tipos de lista que podem dar ainda mais flexibilidade a nossas aplicações. É o que veremos a seguir.

Outros tipos de lista

“ A unidade é a variedade, e a variedade na unidade é a lei suprema do universo. ”

ISAAC NEWTON

Podemos fazer pequenas variações nas listas ligadas e, com essas modificações, deixá-las mais adequadas para inúmeras aplicações.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer outros tipos de listas ligadas;
- diferenciar o uso e a adequação dos tipos de listas para a implementação de aplicações computacionais;
- implementar e manipular os diferentes tipos de lista.



Para começar

Uma sequência de itens pode ser arranjada de diversas formas.



Em nosso dia a dia, muitas vezes nos deparamos com várias listas: de compras, de coisas a fazer, de presentes, de amigos... Essas listas são, quase sempre, sequências de itens semelhantes.

Em sistemas computacionais, além de armazenar conteúdos de diversos tipos, podemos escolher diferentes tipos de lista conforme nossas necessidades e as necessidades de nossos sistemas.

Por serem de vários tipos, as listas podem servir a diversos tipos de aplicação e variar em diferentes aspectos, como:

- **tipo de informação** – as listas podem armazenar diferentes tipos de informação – nomes de amigos, itens de compras e todos os outros elementos que existem no mundo real e desejamos representar nos sistemas computacionais;
- **ordem dos elementos** – os elementos das listas podem ser armazenados segundo diferentes políticas – do mais recente para o mais antigo ou vice-versa, em ordem alfabética, em ordem crescente ou decrescente de código etc.;
- **homogeneidade da informação** – todos os elementos de uma lista podem ser do mesmo tipo (por exemplo, um registro que armazena informações sobre clientes) ou ter tipos diferentes.

Assim, além das listas que vimos no Capítulo 7 (algumas vezes, vamos chamá-las de *listas ligadas simples*), podemos ter listas duplamente ligadas, listas com cabeça fixa, listas circulares, listas multidimensionais e listas transversais, além das combinações entre esses tipos.

Vamos aproveitar nosso conhecimento sobre listas ligadas para facilitar o aprendizado sobre esses novos tipos de lista. Depois, vamos usar esse conhecimento para aprender outras estruturas de dados, como *filas* e *pilhas*.



Conhecendo a teoria para programar

Como vimos no Capítulo 7, podemos armazenar quaisquer informações sobre clientes, fornecedores, produtos, amigos etc. em uma lista ligada, sem precisar definir *a priori* o número máximo de registros dessa lista.

Para acessar os registros (ou nós) da lista, armazenamos em cada nó o endereço do seguinte. Para isso, acrescentamos ao registro de dados um campo ponteiro e, para acessar o primeiro registro, mantemos apenas uma variável do tipo ponteiro na função principal. Esse ponteiro armazena o endereço do primeiro nó da lista.

A seguir, veremos as principais variações na estrutura ou na manipulação das listas e analisaremos quando usar cada uma delas.

Listas duplamente ligadas

Em cada nó de uma lista duplamente ligada (ou *lista duplamente encadeada*), armazenamos o endereço do nó seguinte e também o endereço do nó anterior.

Isso permite:

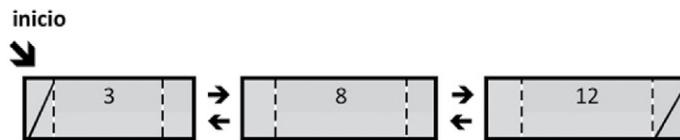
- percorrer a lista nos dois sentidos (do começo até o final, e do final até o começo);
- inserir e remover elementos em ordem (alfabética, crescente ou decrescente) com mais facilidade, pois não precisaremos mais guardar o endereço do nó anterior enquanto percorremos a lista para fazer as ligações de ponteiros corretamente.

Vamos modificar o nosso nó, chamando o novo ponteiro de *ant*, para lembrarmos que ele armazena o endereço do nó anterior (Código 8.1).

Código 8.1

```
typedef struct tipo_no no;  
  
struct tipo_no {  
    int info;  
    struct tipo_no *prox, *ant;  
};
```

Na lista duplamente ligada a seguir, o nó 3 é o primeiro nó da lista (*inicio*). Ele não possui anterior, portanto, seu ponteiro *ant* é NULL. Seu ponteiro *prox* contém o endereço do nó 8. O nó 8 possui nós antecessor e sucessor. Seu campo *ant* contém o endereço do nó 3, e o campo *prox* contém o endereço do nó 12. O nó 12 possui apenas nó anterior (o nó 8). Como não tem sucessor, seu campo *prox* é NULL.



A inicialização da lista duplamente ligada é feita da mesma forma que a de uma lista simples: inicializando o ponteiro para o primeiro elemento com o valor NULL.

Inserção de um nó em uma lista em ordem crescente

Para inserir na lista um novo nó com valor n , em ordem crescente, vamos primeiro alocar espaço para o novo elemento e, depois, ligá-lo à lista. Há várias formas de fazer isso, mas vamos dividir o algoritmo no caso mais simples (lista vazia) e no caso geral (lista não vazia). Caso a lista esteja vazia, o novo nó passará a ser o único elemento da lista. Senão, vamos inseri-lo como primeiro elemento. O nó que estava no início passará a ser o segundo elemento, e assim por diante, mas não será necessário deslocar os nós fisicamente. O Código 8.2 mostra essa função.

Código 8.2

```
void insere_em_ordem (int n, no **inicio)
{
    no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
    if (aux) // conseguiu alocar espaço
    {
        aux->info = n;
        if (!(*inicio) || (n < (*inicio)->info)) // lista vazia ou n é menor que o primeiro elemento
        {
            (aux)->prox = (*inicio); // preenche os ponteiros do novo nó
            (aux)->ant = NULL;
            if (*inicio) (*inicio)->ant = aux; // liga o primeiro nó ao novo nó
            (*inicio) = aux; // o primeiro nó passa a ser aux
        }
        else // procura o local de inserção com o ponteiro auxiliar p
        {
            no *p = (*inicio);
            while ((p->prox != NULL) && (n > p->info)) p = p->prox;
            if (p->prox == NULL) // insere no final da lista
            {
                p->prox = aux;
                aux->prox = NULL;
                aux->ant = p;
            }
            else // insere no meio da lista, após o ponteiro p
            {
                aux->ant = p;
                aux->prox = p->prox;
                p->prox->ant = aux;
                p->prox = aux;
            }
        }
    }
    else printf ("Heap overflow\n");
}
```

Impressão de todos os nós da lista

Algoritmos que não alteram a lista, como os de percorrer a lista e encontrar elementos, podem permanecer os mesmos das listas simples. Veja o Código 8.3.

Código 8.3

```
void imprime (no *inicio)
{
    while (inicio)    // enquanto houver lista
    {
        printf("%d ", inicio->info);    // imprime o valor do nó
        inicio=inicio->prox;            // anda para o próximo nó
    }
}
```

Listas com cabeça fixa

No início de boa parte dos algoritmos de manipulação de listas ligadas, verificamos se a lista está vazia. Se estiver vazia, realizamos um procedimento para cuidar dessa exceção (dando uma mensagem para o usuário ou retornando à função principal, por exemplo). Depois, tratamos o caso geral, que é o da lista não vazia.

Para deixar o código mais simples e compacto, podemos usar listas com cabeça fixa. A única diferença delas é que o primeiro nó é fixo (e denominado *cabeça da lista*) e não armazena nenhum elemento.

Para inicializarmos uma lista com cabeça fixa, alocamos um nó e preenchemos seu campo *prox* com NULL. Os campos de informação não precisam ser preenchidos, pois só fazem parte da lista física (e não da lista lógica). Também podemos usá-los para armazenar informações de gerenciamento, como o número de nós da lista.

Dica

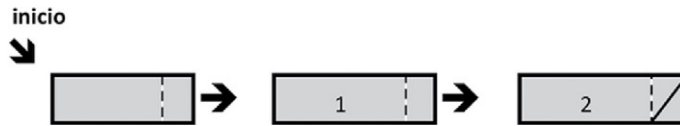
É possível misturar os vários tipos de lista conforme nossa necessidade.

Assim, podemos usar listas duplamente ligadas com cabeça fixa. Hibridizações como essa apresentam características dos tipos de lista que as compõem.

Na inicialização de uma lista duplamente ligada com cabeça fixa, além do ponteiro *prox*, o ponteiro *ant* também deve ser inicializado com NULL.



A seguir, podemos visualizar uma lista com cabeça fixa com dois elementos: um com valor 1 e outro com valor 2.



Inserção de um nó no início da lista

Novamente, para inserir um novo nó com valor n no início da lista, vamos primeiro alocar espaço para o novo elemento e, depois, ligá-lo a ela. Vamos chamar o novo nó de *aux*. Preenchemos o campo de informação e, depois, o campo *prox*. O nó que ficará após o nó *aux* é o primeiro da lista, ou seja, *inicio*->*prox* (lembre-se de que o nó *inicio* é apenas a cabeça da lista, não fazendo parte da lista lógica). Depois, fazemos com que o novo nó (*aux*) passe a ser o primeiro elemento da lista. Para isso, ligamos o campo *inicio*->*prox* a ele.

Código 8.4

```
void insere_inicio (int n, no *inicio)
{
    no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
    if (aux) // conseguiu alocar espaço
    {
        aux->info = n;
        aux->prox = inicio->prox;
        inicio->prox = aux;
    }
    else printf ("Heap overflow\n");
}
```

Compare essa função, implementada no Código 8.4, com a função que insere um nó no final de uma lista ligada simples (ver Capítulo 7). Ficou mais simples, não é?



Dica

Como nenhuma função vai alterar o primeiro nó da lista, dado que esse nó é fixo, não precisamos passar o início da lista por referência. Podemos percorrê-la utilizando esse ponteiro (*inicio*). Como sabemos, quando executamos uma função, são feitas cópias de seus parâmetros, e vamos alterar apenas essa cópia.

Inserção de um nó no final da lista

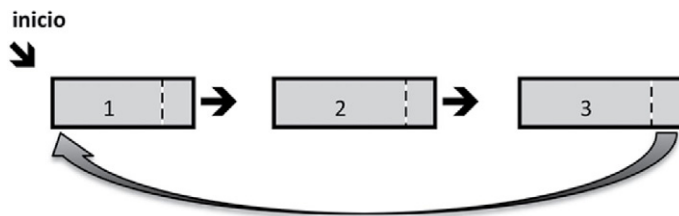
Agora, vamos ver como faríamos para inserir um novo nó no final da lista. Para isso, percorremos a lista até encontrar o último nó. Então, ligamos seu campo *prox* ao novo nó. Como em qualquer lista com cabeça fixa, não precisamos testar se ela está vazia, o que simplifica o algoritmo (Código 8.5).

Código 8.5

```
void insere_fim (int n, no *inicio)
{
    no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
    if (aux) // conseguiu alocar espaço
    {
        aux->info = n;
        aux->prox = NULL; // como o novo nó será inserido no final da lista, ele não terá ninguém à frente
        while (inicio->prox != NULL)
            inicio = inicio->prox;
        inicio->prox = aux;
    }
    else printf ("Heap overflow\n");
}
```

Listas circulares

O que difere as listas circulares das outras é que nelas o último nó aponta para o primeiro, ou seja, o campo *prox* do último nó contém o endereço do primeiro nó. Observe a figura a seguir:



As listas circulares são usadas para manipular eventos cíclicos, tais como:

- se os nós armazenarem endereços de servidores, podemos fazer um programa que envie requisições de clientes para determinado servidor, depois para o *proximo*, depois para o *proximo*, e assim por diante. Dessa forma, conseguimos balancear a carga dos servidores, isto é, dividi-la, sem sobrecarregar nenhum servidor;

- se os nós representarem processos (programas sendo executados por um sistema operacional), o escalonador, que é a parte do sistema operacional que escolhe qual processo vai executar em determinado *core*, pode usar uma fila circular para executar vários processos, cada um por um pequeno intervalo de tempo. Se todos os processos forem executados rapidamente (em até 0.2 segundos, considerado o tempo médio de reação de um ser humano), o usuário terá a impressão de que os processos estão sendo executados ao mesmo tempo, em paralelo.

A manipulação de listas circulares é muito parecida com a das listas simples. A única diferença é a forma de reconhecer que a lista chegou ao fim. Você consegue imaginar como se pode fazer isso? Observe o último nó da lista circular.

Isso mesmo! O campo *prox* do último nó armazena o endereço do *inicio* em vez de armazenar *NULL*. A seguir veremos alguns exemplos.

Impressão de todos os nós da lista

Para imprimir todos os nós da lista, primeiro testamos se ela não está vazia (poderíamos não precisar testar isso, se usássemos uma lista circular com cabeça fixa). Se a lista não estiver vazia, devemos percorrê-la usando um ponteiro auxiliar, começando no início da lista e imprimindo todos os elementos até que ele chegue novamente ao início, já que a lista é circular (Código 8.6).

Código 8.6

```
void imprime (no *inicio)
{
    if (inicio)                                // se a lista não for vazia
    { no *aux = inicio;                          // aux começa no início da lista
      do{
          printf("%d ",aux->info);              // imprime o valor do nó
          aux=aux->prox;                        // anda para o próximo nó
      } while (aux != inicio);                  // até aux dar a volta na lista
    }
}
```

Remoção do primeiro nó da lista

A remoção do primeiro nó de uma lista circular é um pouco mais complicada, porque precisamos encontrar o último nó e ligar seu campo *prox* ao segundo nó. Só então poderemos ligar o último nó ao segundo e fazer com que ele se torne o primeiro nó da lista. Mas, antes disso, testamos se a lista possui apenas um nó; nesse caso, temos que atribuir o valor *NULL* a ela, a fim de que as outras funções possam saber que a lista está vazia. O Código 8.7 mostra a função.

Código 8.7

```

int remove_inicio (no **inicio)
{
    if (!(*inicio)) return (-1); // retorna a flag -1 para informar que a lista estava vazia
    no *aux=(*inicio);           // guarda o primeiro nó em aux
    int n=(*inicio)->info;       // guarda o valor do nó para retornar
    if ((*inicio)->prox == (*inicio) // se houver apenas um nó na lista
        {
            (*inicio) = NULL;       // a lista passa a ser vazia
            free(aux);               // libera o espaço de memória
            return(n);               // retorna o valor do nó removido
        }
    else {                        // lista com mais de um elemento
        while (aux->prox !=(*inicio))
            aux = aux->prox; // achamos o último nó para ligarmos ao primeiro
        aux->prox = (*inicio)->prox; // ligamos o último ao segundo nó
        free(*inicio);
        (*inicio)=aux->prox; // anda com inicio para o segundo nó
        return(n);
    }
}

```

Listas multidimensionais

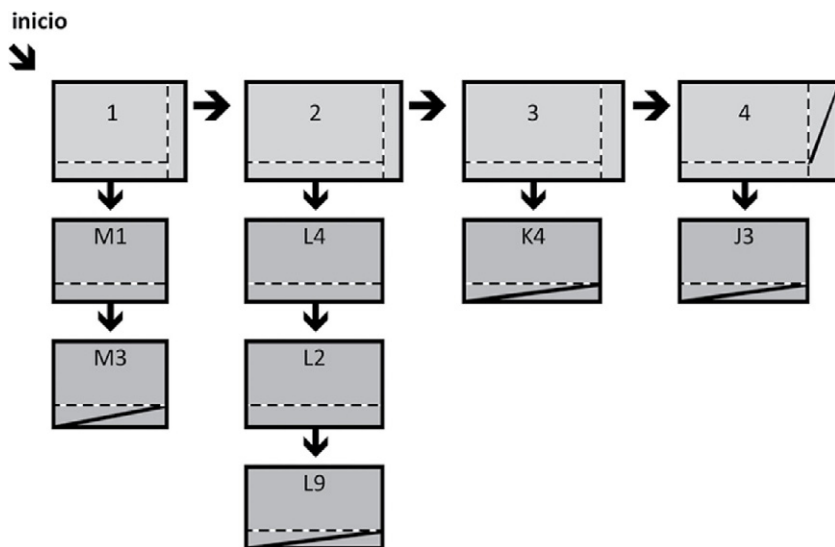
Também é possível (e, às vezes, muito útil) construir e utilizar listas multidimensionais, adequadas para organizar as informações e reduzir o tempo de busca de um registro.

Listas multidimensionais nada mais são do que listas dentro de listas. Para cada dimensão, somamos um ponteiro à nossa estrutura de nó.

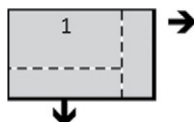
Na a seguir vemos uma lista bidimensional. A lista horizontal, composta pelos nós com valores 1, 2, 3 e 4, poderia ser uma lista de categorias, como nomes de cantores ou bandas em um sistema para gerenciar as músicas que você possui, por exemplo. Claro que o campo *info* dela poderia ser trocado para o tipo *string*, armazenando um nome.

A lista vertical é uma lista de elementos de dados (registros), como os nomes das músicas que você possui, por exemplo. Seu campo de informação está representado como uma *string*. Assim, a lista composta pelos nós M1 e M3 seriam as músicas do tipo 1 (da banda 1), os nós L4, L2 e L9 seriam as músicas do tipo 2, e assim por diante.

Para inserir, remover ou consultar uma música, procuraremos primeiro pelo identificador do cantor/banda, seguindo a lista horizontal, e depois buscaremos a música na lista vertical.



Mas vamos primeiro construir nossa lista, observando os nós da lista horizontal, onde estão as categorias (ou tipos):

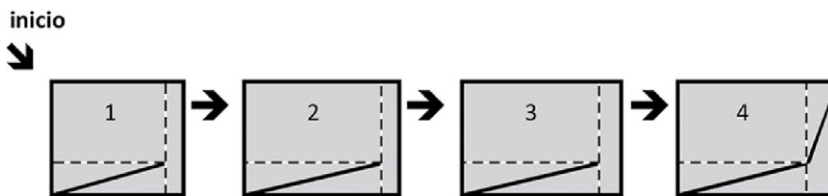


Cada nó é composto por um campo de informação e dois ponteiros. Um dos ponteiros (da lista horizontal) aponta para outro nó, que também armazena um tipo. O outro ponteiro (da lista vertical) aponta para outro tipo de nó, que armazena o dado propriamente dito (a música). Assim, esse nó, mostrado no Código 8.8, tem uma diferença em relação aos nós anteriores: possui um ponteiro para outro tipo de nó, a que chamaremos de *tipo_no2*:

Código 8.8

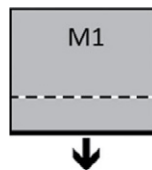
```
typedef struct tipo_no no;
struct tipo_no {
    int info;
    struct tipo_no *prox;
    struct tipo_no2 *ini;
};
```

Para construir a lista horizontal, podemos utilizar os algoritmos anteriores, somente tomando o cuidado de inicializar o ponteiro *ini* (início da lista vertical) com *NULL*. Observe a lista de tipos:



Essa lista é muito parecida com nossas listas anteriores, não é?

A lista vertical também é muito parecida com nossas listas anteriores — ou igual a elas! —, mas, como ela armazena conteúdo diferente, vamos criar outro tipo de nó. Observe a estrutura de cada nó dessa lista:



Esse nó contém um campo de informação (agora *string*), que vamos chamar de *nome*, e um ponteiro para outro nó do mesmo tipo (*tipo_no2*) (Código 8.9).

Código 8.9

```
typedef struct tipo_no2 no2;  
  
struct tipo_no2 {  
    char nome[20];  
    struct tipo_no2 *p;  
};
```



Dica

As listas de tipos funcionam como uma “lista de cabeças”. Assim, cada lista de elementos (vertical) é uma lista com cabeça fixa.

Como dissemos, podemos usar os mesmos algoritmos de manipulação de listas, apenas tomando os seguintes cuidados:

- inicializar o ponteiro para a lista de elementos (*ini*) com *NULL*;
- identificar quando o ponteiro contém um endereço do nó de tipos (*no*) ou do nó de elementos (*no2*). No segundo caso, o nome do ponteiro que armazena o próximo nó foi chamado de *p* (isso foi feito para você identificar os dois tipos de nó, mas, se você quiser facilitar sua programação, pode chamar os dois de *prox*).

Vamos dar alguns exemplos.

Encontrar um elemento da lista de tipos

Para verificar se um tipo de música (por exemplo, de determinada banda) já está cadastrado, podemos usar o algoritmo de busca do Capítulo 7, sem nenhuma alteração, como visto no Código 8.10.

Código 8.10

```
no* busca (int n, no *inicio)
{
    while (inicio)
    {
        if (inicio->info == n) return (inicio); //encontrou o elemento
        inicio=inicio->prox;
    }
    return (NULL); // acabou a lista sem encontrar o elemento.
}
```

Inserção de um tipo no final da lista

Caso tenhamos buscado um tipo de música ainda não cadastrada, poderemos cadastrá-lo no final da lista usando o algoritmo do Capítulo 7, com uma pequena alteração.

Suponhamos que desejemos verificar se o tipo n está cadastrado. Se não estiver, vamos cadastrá-lo no final da lista. O comando seria este:

```
if (busca(n,inicio) == NULL) insere_fim(n,&inicio);
```

A mudança no algoritmo de inserção no final consiste em inicializar o ponteiro para a lista de elementos (*ini*) com *NULL*. A alteração (Código 8.11) está inserida em destaque no código original.

Código 8.11

```
void insere_fim (int n, no **inicio)
{
no* aux = (no*) malloc (sizeof(no)); // aloca espaço para o novo nó
if (aux) // conseguiu alocar espaço
{
    aux->info = n;
    aux->prox = NULL; // como o novo nó será inserido no final da lista, ele não terá ninguém à frente
    aux->ini = NULL; // a lista de elementos começa vazia até que algum elemento seja inserido
    if (!(*inicio)) // lista vazia
        (*inicio) = aux;
    else // lista não vazia
    {
        no *p = (*inicio); // usaremos o ponteiro p para encontrar o último nó
        while (p->prox != NULL)
            p = p->prox;
        p->prox = aux;
    }
}
else printf ("Heap overflow\n");
}
```

Para a lista de elementos, também podemos utilizar os algoritmos anteriores, tomando os mesmos cuidados e lembrando-nos de que se trata de listas com cabeça fixa. Vamos ver alguns exemplos.

Inserção de um elemento no início da lista

Para inserir um novo elemento (uma nova música, por exemplo), podemos, fazer uma busca pelo tipo dele, usando a função *busca*, que retorna o endereço do registro com o valor pedido (*n*). No caso da nossa lista, esse endereço é o da cabeça da lista, que armazenamos na variável *aux*. Em seguida, inserimos a nova música (*nome*) na lista desejada.

Código 8.12

```
no *aux;
aux=busca(n,inicio;
insere_inicio(nome, aux);
```

Nessa chamada, mostrada no Código 8.12, consideramos que o tipo *n* já está cadastrado (se não estiver, podemos usar o algoritmo de inserção de um tipo no final da lista de tipos que vimos anteriormente).

A função de inserção no início da lista possui a estrutura já vista. As alterações feitas abaixo referem-se ao campo de informação (agora uma *string* chamada *nome*), o tipo do nó da lista vertical (*no2*) e os nomes dos ponteiros (o da cabeça da lista para o início da lista de elementos chama-se *ini*, e o ponteiro de próximo chama-se *p*). Vejamos no Código 8.13 como ficou.

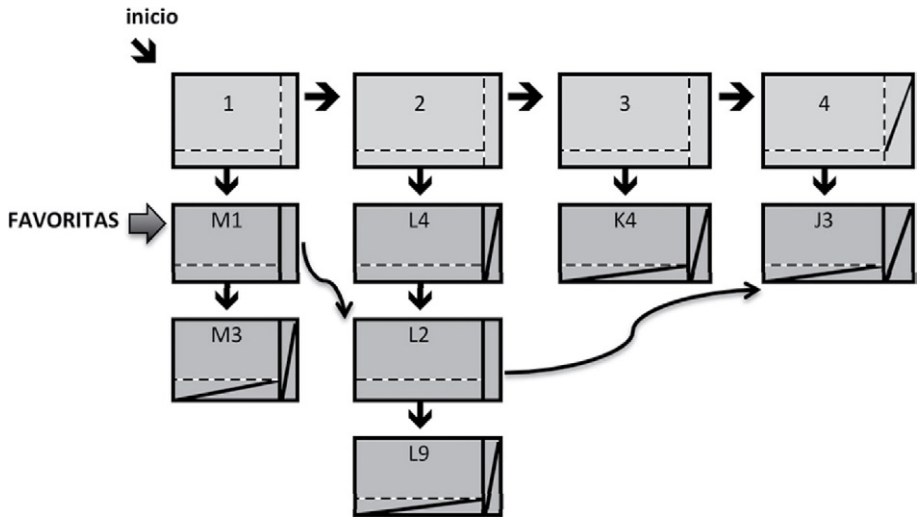
Código 8.13

```
void insere_inicio (char *nome, no2 *ini)
{
no2* aux = (no2*) malloc (sizeof(no2)); // aloca espaço para o novo nó
if (aux) // conseguiu alocar espaço
{ strcpy(nome, aux->nome);
  aux->p = ini->p;
  ini->p = aux;
}
else printf ("Heap overflow\n");
}
```

Listas transversais

Em todos os tipos de lista, podemos inserir outras listas, chamadas de *transversais* ou *costuradas*. Assim como nas listas multidimensionais, cada lista transversal exige um ponteiro adicional na estrutura do nó.

No exemplo anterior, podemos criar uma lista transversal, por exemplo com as músicas favoritas, composta pelas músicas M1, L2 e J3.



Essa lista terá um ponteiro chamado *FAVORITAS* declarado e inicializado com *NULL*, assim como já havíamos feito com o ponteiro *inicio*. Como esse ponteiro conterà o endereço de um elemento, ele será do tipo *no2 **. A lista de músicas favoritas será ligada por meio do ponteiro *next*, que armazenará o endereço da próxima música favorita. Vejamos no Código 8.14 a alteração no registro:

Código 8.14

```
typedef struct tipo_no2 no2;

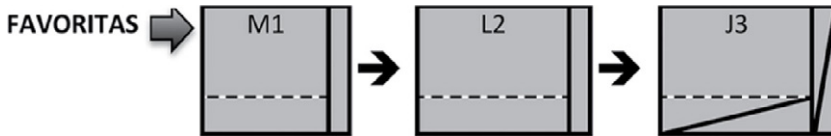
struct tipo_no2 {
    char nome[20];
    struct tipo_no2 *p;
    struct tipo_no2 *next;
};
```

No Código 8.15 vemos a declaração e a inicialização do ponteiro *FAVORITAS*:

Código 8.15

```
no2 * FAVORITAS = NULL;
```

Como a lista de músicas favoritas nada mais é do que uma lista simples, podemos continuar usando nossos algoritmos para inserir nós, consultar e imprimir todas as músicas favoritas, por exemplo. Observe:



Vamos programar

Como vimos, a programação em listas multidimensionais e transversais é muito similar à programação nas listas lineares. Assim, mostraremos os algoritmos para listas lineares (Códigos 8.1 a 8.7).

Java

Código 8.1

```
public class No {  
  
    public int codigo;  
    public No prox;  
    public No ant;  
  
}
```

Código 8.2

```
public static void insere_em_ordem(int n, LinkedList<Integer> list){
    int i=0;
    if(list.isEmpty()){ // Se a lista é vazia, insere na primeira posição
        list.add(n);
    } else {

        while(i < list.size()){ // Percorre até o tamanho máximo da lista
            if (i >= (list.size())){
                list.addLast(n); // Insere na ultima posição
                break;
            }
            if (list.get(i).intValue() > n){
                list.add(i, n); // Insere na posicao correta
                break;
            }
            i++;
        }
    }
}
```

Código 8.3

```
System.out.println("LinkedList: " + list); // Imprime a lista
```

Código 8.4

```
List.addFirst(Elemento); // Insere no Inicio
```

Código 8.5

```
List.addLast(Elemento); // Insere na Ultima posição
```

Código 8.6

```
List.removeFirst(); // Remove o primeiro elemento
```

Código 8.7

```
class No :  
    def __init__(self, info=None, prox=None, ant=None):  
        self.info = info  
        self.prox = prox  
        self.ant = ant  
    #toString()  
    def __str__(self):  
        return str('Info:%s\n' %(self.info))
```

Python

Código 8.1

```
class No :  
    def __init__(self, info=None, prox=None, ant=None):  
        self.info = info  
        self.prox = prox  
        self.ant = ant  
    #toString()  
    def __str__(self):  
        return str('Info:%s\n' %(self.info))
```

Código 8.2

```
class ListaDupla:
    def __init__(self):
        self.inicio = None

    def insere_em_ordem(self, n):
        aux = No() #aloca espaço para o novo nó
        aux.info = n
        aux.prox = None

        if self.inicio == None or n < self.inicio.info: #lista vazia ou n é menor que o primeiro elemento
            aux.prox = self.inicio #preenche os ponteiros do novo nó
            aux.ant = None
            self.inicio = aux #o primeiro não passa a ser aux
        else: #procura o local de inserção como o ponteiro auxiliar p
            p = self.inicio
            while p.prox != None and n > p.prox.info:
                p = p.prox
            if p.prox == None: #insere no final da lista
                aux.prox = None
                aux.ant = p
                p.prox = aux
            else: #insere no meio da lista, após o ponteiro p
                aux.ant = p
                aux.prox = p.prox
                p.prox.ant = aux
                p.prox = aux
```

Código 8.3

```
def imprime(self):
    print "Elementos da lista: "
    aux = self.inicio
    while(aux): #enquanto houver lista
        print aux.info #imprime o valor do nó
        aux = aux.prox #anda para o próximo nó
```


Código 8.4

```
def insere_inicio(self, num):  
    aux = No() #aloca espaço para o novo no  
    aux.info = num  
    if self.inicio == None:  
        self.inicio = aux  
    else:  
        aux.prox = self.inicio.prox  
        self.inicio.prox = aux
```

Código 8.5

```
def insere_fim(self, n):  
    aux = No() #aloca espaço para o novo no  
    aux.info = n  
    aux.prox = None #como o novo no será inserido no final da lista, ele não terá ninguém a frente  
    if self.inicio == None:  
        self.inicio = aux  
    else:  
        p = self.inicio  
        while(p.prox):  
            p = p.prox  
        p.prox = aux
```

Código 8.6

```
class ListaCircular:  
    def __init__(self):  
        self.inicio = None  
  
    def imprime(self):  
        print "Elementos da lista: "  
        if self.inicio <> None: #se a lista não for vazia  
            aux = self.inicio #aux começa no início da lista  
            print aux.info #imprime o valor do nó  
            aux = aux.prox #anda para o próximo nó  
            while (aux <> self.inicio): #até aux dar a volta na lista  
                print aux.info #imprime o valor do nó  
                aux = aux.prox #anda para o próximo nó
```

Código 8.7

```
def remove_inicio(self):
    if self.inicio==None: #retorna a flag -1 para informar que a lista estava vazia
        return -1
    aux=self.inicio #guarda o primeiro nó em aux
    n=self.inicio.info #guarda o valor do nó para retornar
    if self.inicio.prox==self.inicio: #se houver apenas um nó na lista
        self.inicio=None #a lista passa a ser vazia
    else: #lista com mais de um elemento
        while aux.prox <> self.inicio: #achamos o último nó para ligarmos ao primeiro
            aux=aux.prox
        aux.prox=self.inicio.prox #ligamos o último ao segundo nó
        self.inicio=aux.prox #anda com início para o segundo nó
    return n #retorna o valor do nó removido
```



Para fixar

Implemente as funções a seguir para listas duplamente ligadas com elementos ordenados em ordem crescente:

1. função que remove o elemento de valor n da lista;
2. função que imprime todos os elementos da lista em ordem decrescente;
3. função que recebe um parâmetro inteiro n e retorna a quantidade de elementos com valores iguais ou maiores que n .



Para saber mais

Assim como para as listas ligadas simples, você pode ter mais exemplos de outros tipos de listas no Capítulo 4 do livro *Algoritmos e estruturas de dados* (Wirth, 1999).



Navegar é preciso

Vários algoritmos para a manipulação de registros em listas duplamente ligadas e ordenadas podem ser encontrados em:

<http://www.mat.uc.pt/~amca/ED0506/FinalListas.pdf>. Por meio do link <http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html>, você poderá obter outros exemplos de algoritmos para manipulação de listas simples e com cabeça fixa, por exemplo.

Exercícios

Complete o exemplo de uso das listas transversais e implemente:

1. uma função que busque um elemento (uma música) e a insira na lista de músicas favoritas;
2. uma função que imprima quais músicas favoritas são de determinado tipo (cantor ou banda).

Referência bibliográfica

WIRTH, N. *Algoritmos e estruturas de dados*. São Paulo: LTC, 1999.



QUE VEM DEPOIS

As listas ligadas são estruturas úteis e muito usadas. Existem duas outras estruturas de dados que poderíamos considerar casos particulares das listas ligadas, mas, por terem políticas próprias de inserção e remoção de nós, além de muitas aplicações, são consideradas novas estruturas de dados – *filas e pilhas*.

Filas

“ A fila anda. ”

AUTOR DESCONHECIDO

Quem nunca se deparou com essa frase, principalmente depois de uma desilusão amorosa? Sua função básica é nos deixar com a ideia de que algo se foi e outra coisa está chegando. Neste capítulo, veremos claramente que a fila anda, e as informações também.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- identificar situações e maneiras de utilizar filas;
- simular situações do mundo real em um ambiente virtual;
- entender como vários recursos computacionais utilizam-se das filas para controlar a chegada e o consumo de dados.



Para começar

Como sabemos, hoje a população mundial ultrapassa os 7,2 bilhões de habitantes e, se não mantivéssemos um mínimo de respeito, ética e ordem, nossa convivência não seria possível, muito menos teríamos a sociedade que conhecemos, com suas virtudes e problemas.



Nesse cenário populacional e social que naturalmente vivenciamos, temos inúmeras regras e costumes que nos fazem conviver com harmonia. Certamente, uma das principais regras é sermos justos uns com os outros. Imagine alguém sendo injusto com você. É agradável? Imagino que não.

Imagine que você precise de um serviço ou recurso que não esteja numericamente à disposição de todos ao mesmo tempo, mas tenha tempo de utilização limitado, como um caixa de banco, por exemplo. Naturalmente, esperamos o término da utilização por outra pessoa para dar início à nossa. Agora pondere: por que você é o próximo a utilizar esse serviço? Não seria a vez de outra pessoa? Pois então: existe uma ordem predefinida, que todos nós sabemos bem como funciona, ou seja, quando não há uma regra determinada para a utilização de um serviço utilizamos a ordem de chegada. Mas por que temos essa regra? Sim, foi isso mesmo que você pensou: para sermos justos uns com os outros. Você teria um sentimento de injustiça se outras pessoas, que chegaram depois de você, tivessem acesso ao serviço antes.

Essa regra força as pessoas a se enfileirar para ter acesso ao serviço, pois é intrínseco entender que quem chegou antes terá acesso ao serviço primeiro e, obviamente, quem chegou depois terá acesso depois, portanto, não seria justo que acessasse o serviço antes de você, ou seria? É claro que não.

Gostaríamos de exemplificar utilizando fatos corriqueiros com que, provavelmente, a grande maioria de nós se depara. Será que você poderia nos ajudar? Temos certeza de que sim, vamos lá!

Imagine que você e seus colegas de sala de aula saíram de uma maravilhosa e empolgante aula de Estrutura de Dados e estão com muita fome – afinal, é hora do almoço! Entretanto, como o professor se empolgou em responder às dúvidas geradas pelos exercícios dados no final da aula, já é um pouco mais tarde do que de costume e parece que todas as outras turmas resolveram chegar minutos antes de vocês à praça de alimentação. Resultado: todos os restaurantes estão cheios. E então, qual é a sua primeira reação? Imaginamos que seria filtrar os restaurantes de sua preferência; depois, analisar o tamanho da fila, mesmo porque isso lhe mostraria quantas pessoas seriam atendidas antes de você, já que, em filas, o primeiro a chegar é o primeiro a ser atendido, correto?

Atenção:



Nas filas, o primeiro a chegar é o primeiro a ser atendido. No mundo virtual não há como ter penetras, a não ser que você mesmo implemente tal funcionalidade.

Assim como na vida real, podemos ter filas com diferentes prioridades de atendimento, assim como temos filas para idosos e deficientes.

Já temos todos os conhecimentos a respeito de como funciona uma fila. Agora precisamos fazer com que o conceito do mundo real seja traduzido para o mundo virtual. É isso o que faremos neste capítulo.



Papo técnico:

Temos vários recursos computacionais que são compartilhados, como processador e *sockets*, e usamos os conceitos de fila para controlar sua utilização.



Conhecendo a teoria para programar

Acabamos de perceber que temos todos os conceitos necessários para entender uma fila, mas, para traduzi-la de forma adequada para o mundo virtual, precisamos identificar detalhes pequenos porém de extrema importância para uma perfeita implementação, como o início e o fim da fila, além de perceber que, no mundo real, quem entra na fila é um ser humano, ao passo que no mundo virtual dizemos, genericamente, que “colocamos informações na fila”.



Papo técnico:

A “informação colocada na fila” pode ser qualquer tipo primitivo disponível ou abstrato criado no seu programa.

Precisamos controlar adequadamente as extremidades das filas, pois será nelas que faremos as manipulações. Imagine que você está chegando à fila do restaurante para comer. Onde você se posiciona? No fim da fila, certo? De outro lado, quando você estiver na iminência de ser atendido, estará no início da fila.



Conceito:

Toda informação que chega à fila é adicionada no FIM; toda informação a ser consumida pelo recurso é retirada do INÍCIO da fila. Lembre-se: você SEMPRE entra no fim da fila.

A [Figura 9.1a](#) ilustra uma fila com três elementos, A, B e C, em que o elemento A é o início da fila e o elemento C é o fim. Nesse cenário, caso venhamos a tirar um elemento da fila, de acordo com o que já discutimos, o que estiver no início é o que deverá ser removido, resultando na [Figura 9.1b](#).

E se solicitássemos a você que incluísse um novo elemento, onde você o colocaria? Isso mesmo: você o colocaria no fim da fila, por isso apresentamos a sua solução na [Figura 9.1c](#). Neste momento, deve estar bem claro que B seria o próximo elemento a ser removido, depois C e, posteriormente, D.

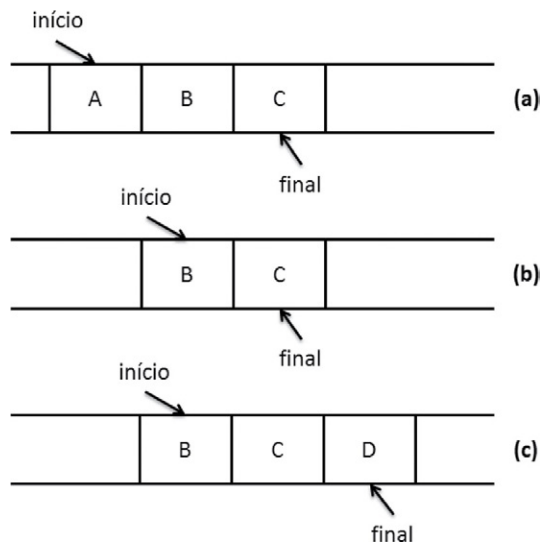


FIGURA 9.1: Uma fila.



Papo técnico:

As filas são comumente chamadas de *filas fifo* (*first-in, first-out* – o primeiro a entrar é o primeiro a sair).

Agora falemos um pouco sobre como esses conceitos de *fila* podem ser usados em programação. Depois de ter estudado, em capítulos anteriores, vários conceitos e técnicas interessantes de guardar e manipular informação, como você imagina que possamos implementar uma fila? Existem duas formas principais. Será que você imaginou as duas? Esperamos que sim!

Se os conceitos de *lista ligada* apresentados nos capítulos anteriores ainda estiverem frescos em sua mente, provavelmente você imaginou implementar uma fila usando os nós da lista ligada. Isso mesmo, essa é uma das formas. A outra é mais simples e exige um conceito menos avançado de programação, que é a utilização de vetor.

Independentemente de como a implementação será feita, para manipular uma fila teremos duas funcionalidades características, que, posteriormente, poderão ser utilizadas com a implementação com vetor e a lista ligada.

Retomando o exemplo da fila do restaurante da praça de alimentação, que funcionalidades você enxerga na manipulação dessa fila? É exatamente isso que você pensou: *entrar na fila* e *sair da fila*.

Para entrar na fila, você deve se posicionar imediatamente atrás da última pessoa que está nela e, nesse instante, se tornará o último da fila, posição que será procurada pelo próximo a entrar no fim da fila. De outro lado, se você for o primeiro da fila, quando o caixa for liberado e chamar o próximo cliente, você sairá da fila e será atendido. Portanto, a pessoa que estiver atrás de você se tornará, naquele momento, o primeiro da fila, sendo a próxima a ser chamada (veja o exemplo dessa dinâmica na [Figura 9.2](#)). Isso também acontecerá em programação. Como é que você pensa em fazer isso? Pense um pouco antes de continuar lendo este texto.

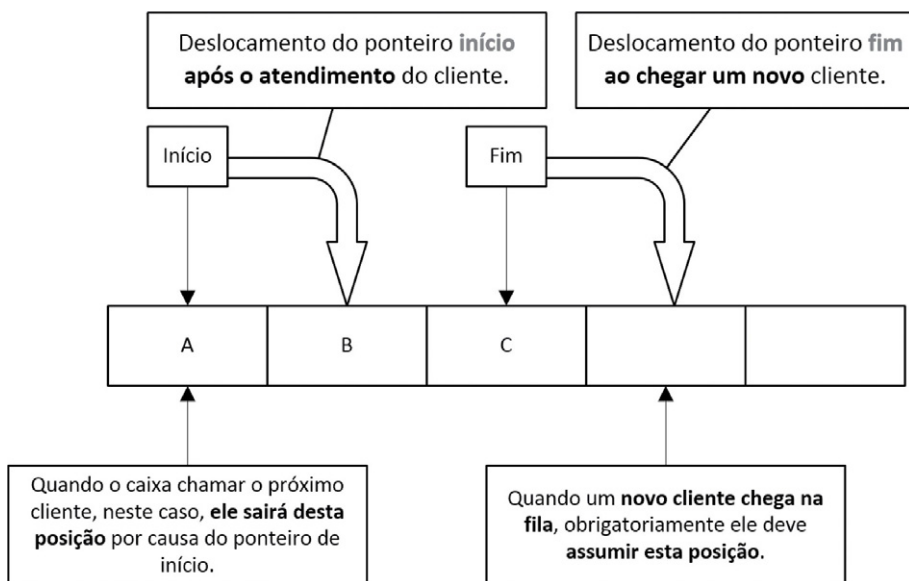


FIGURA 9.2: Exemplificação gráfica da entrada e da saída de um cliente em uma fila.

Levando em consideração o que foi descrito no parágrafo anterior, fica claro que seria conveniente elaborar duas funções para manipular a pilha: uma para inserir nova informação e outra para consumir uma informação já presente nela. Chamaremos essas funções de *inserir* e *consumir*, respectivamente.

Nessa implementação usaremos um vetor como armazenamento das informações em forma de fila. Para isso, precisamos ter alguns cuidados e definições:

- estipular um tamanho para o vetor (quantidade máxima de informações guardadas nele);
- verificar se não chegamos ao final do vetor, ou seja, se não o *estouramos*;
- ter duas variáveis *início* e *fim* para controlar as posições do vetor onde estão essas posições;

- a posição que a variável *inicio* aponta tem a informação para ser consumida, assim como a variável *fim* aponta para a última posição que tem informação armazenada.

Para essa implementação, o que você acha de definir como será a nossa fila? Então, vamos lá!

Primeiramente, é necessário definir o tamanho da fila – no caso deste exemplo, a quantidade de informações que podem passar pela fila não significa, necessariamente, que seja o tamanho máximo que ela pode assumir durante a execução. Ela poderá ter esse tamanho única e exclusivamente se não houver consumo de informação; quando chegarem as informações nessa quantidade, a fila ficará cheia.

```
#define TAMANHO 100
```

Com essa definição podemos criar uma estrutura de dados abstrata, que representará nossa fila. Nela teremos a fila, que nada mais é que um vetor de números inteiros e duas variáveis inteiras, tendo como valor armazenado o índice do vetor que representa a posição onde estão o início e o fim da fila.

Depois dessa explicação, será que você conseguiria montar sua própria estrutura? Temos certeza de que sim. Só para que você confira e nós possamos continuar a explicação, faremos a definição da nossa sugestão de estrutura nas próximas linhas. Além disso, daremos, na [Figura 9.3](#), uma representação gráfica dessa estrutura. Observe:

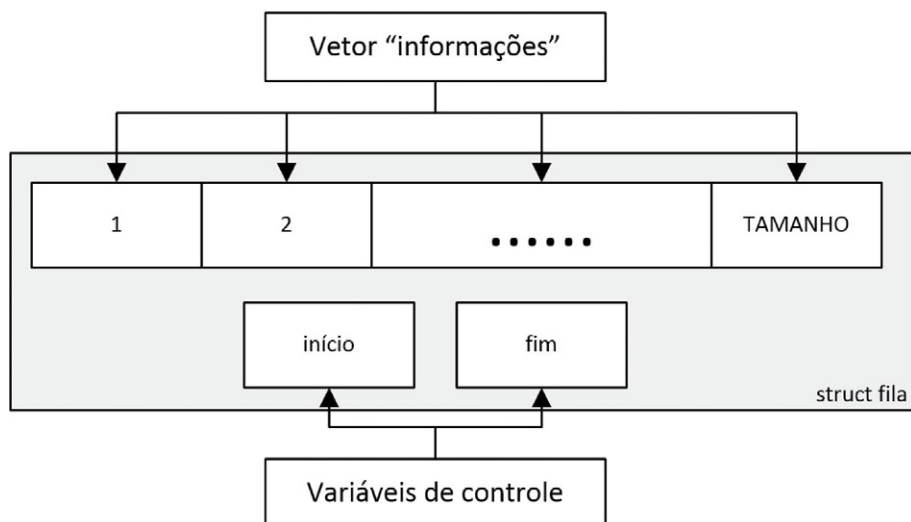


FIGURA 9.3: Exemplificação gráfica da estrutura definida para a fila.

```
#define TAMANHO 100 //definição de uma constante para o tamanho do vetor
struct fila {          // criando uma estrutura
    int informações[TAMANHO]; // vetor que controla a fila
    int inicio, fim;      // variáveis que controlam as posições de início e fim da fila dentro do vetor
};
```

Agora que temos nossa fila definida, o que podemos fazer? Vamos elaborar as funções *inserir* e *consumir* uma informação? Então, vamos lá. Vamos começar com a função *inserir*.

Vale ressaltar que a função para inserir um novo elemento recebe o endereço de memória onde se encontra a fila definida dentro da função que a chamou, nesse caso, provavelmente a função principal; o outro parâmetro é o valor com que faremos a inserção dentro da fila.

```
void inserir(struct fila *f, int valor) {
```

Para *inserir* adequadamente nova informação na fila, temos de verificar se existe a próxima posição que usaremos com uma informação nova. Vale lembrar que a variável que guarda a posição final está com a posição do vetor onde tivemos a última inserção de valor, ou seja, a próxima posição do vetor é que deve existir para receber a próxima informação. Lembre-se do exemplo do restaurante: ao chegar à fila, você procura a última pessoa e se acomoda imediatamente atrás dela; portanto, no vetor, a variável guarda a última posição preenchida, e a próxima informação vem na posição posterior, que deve existir, por isso a verificação. Observe, na [Figura 9.4](#), o posicionamento do ponteiro *fim*, que está na última posição possível, para inserir nova informação. Essa verificação é uma simples condicional. Observe:

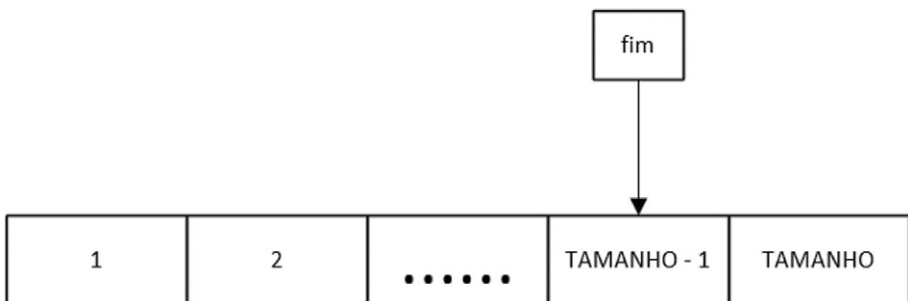


FIGURA 9.4: Posição da última possibilidade de inserir novo elemento.

```
if ((f->fim + 1) < TAMANHO) {
```

Para fazer a inserção da informação, devemos apenas atualizar a posição que a receberá e, simplesmente, atribuí-la. Isso é feito com as próximas duas linhas de código e demonstrada na [Figura 9.5](#):

```
f->fim++; // atualiza a posição final para inserir um novo valor
f->informacoes[f->fim] = valor; // insere o valor na posição
```

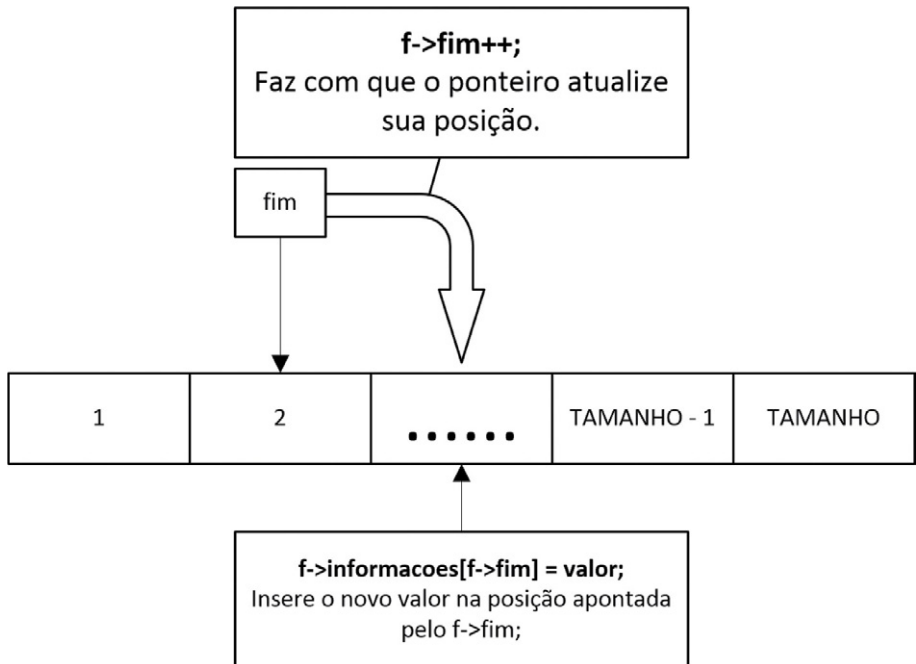


FIGURA 9.5: Dinâmica de uma inserção na fila.

A seguir, apresentamos a sugestão completa do código para essa função.

Código 9.1

```
// Função para inserir um novo elemento na fila
void inserir(struct fila *f, int valor) { // recebe o endereço da fila e o valor que será colocado
    if ((f->fim + 1) < TAMANHO) // verifica se não chegamos ao final da fila
    {
        f->fim++; // atualiza a posição final para inserir um novo valor
        f->informações[f->fim] = valor; // insere o valor na posição
        printf(">>> Inserido >>>: %d\n\n", valor); // imprimi na tela para controle
    }
    else
        printf("Estouro de fila\n\n"); // avisa que houve estouro de fila
}
```

O passo seguinte é apresentar o código proposto para a função *consumir*. Vale ressaltar que ela é um pouco mais complexa, por causa de suas verificações, porém, repare que o conceito principal é o mesmo que já discutimos amplamente nos parágrafos anteriores.

Assim como na função *inserir*, essa função também recebe o endereço onde está nossa fila. Você se lembra porque isso precisa ser feito? Obviamente sim! Você acaba de pensar que é para que essa função tenha total acesso à fila criada na função que chamou essa função – nesse caso, ter acesso à informação que está no *início* da fila, bem como consultar e atualizar a variável que controla essa posição inicial. O cabeçalho da função fica assim:

```
int consumir(struct fila *f) { // recebe o endereço da fila
```

Após criar uma variável local para receber o valor que foi consumido da fila, é necessário fazer a primeira verificação, para identificar se ainda temos posição inicial válida, ou seja, se a posição inicial ainda está abaixo da última posição. Por isso, uma comparação entre a posição inicial sendo

menor que o valor da última posição. Veja abaixo nossa sugestão de código, e depois representado visualmente, na [Figura 9.6](#):

```
int valor; // cria uma variável local para receber o valor consumido
if(f->inicio < TAMANHO) { // verifica se não chegamos ao final da fila
```



FIGURA 9.6: Quando o ponteiro *inicio* aponta para a última posição.

Após saber que temos uma posição de início de fila adequado, é importante verificar se temos alguma informação na fila, ou seja, se estivermos com o início da fila na posição inicial e a informação que tivermos for -1, valor que indica que o vetor apenas foi inicializado (veremos essa inicialização na função principal), isso caracterizará que não temos nenhuma informação na fila, portanto, não há o que consumir. Fizemos a verificação com a condicional a seguir, e demonstramos esse cenário visualmente na [Figura 9.7](#):

```
if((f->inicio == 0) && (f->informações[f->inicio] == -1)) {
```



FIGURA 9.7: Cenário em que não houve inserções na fila.

Cair no *else* dessa condição significa que estamos em uma posição que pode ser a inicial, mas tem algo a ser consumido, portanto, é hora de fazer o que estávamos realmente querendo nessa função. Isso acontecerá em apenas duas linhas de código, pois basta guardar a informação que está nessa posição e atualizar a variável que guarda a posição onde está o início da fila. A [Figura 9.8](#) ilustra esses passos, e nossa sugestão de código é apresentada abaixo:

```
valor = f->informacoes[f->inicio]; // obtém o valor da posição e atribui na variável local  
f->inicio++; // atualiza a posição onde se encontra a posição inicial
```

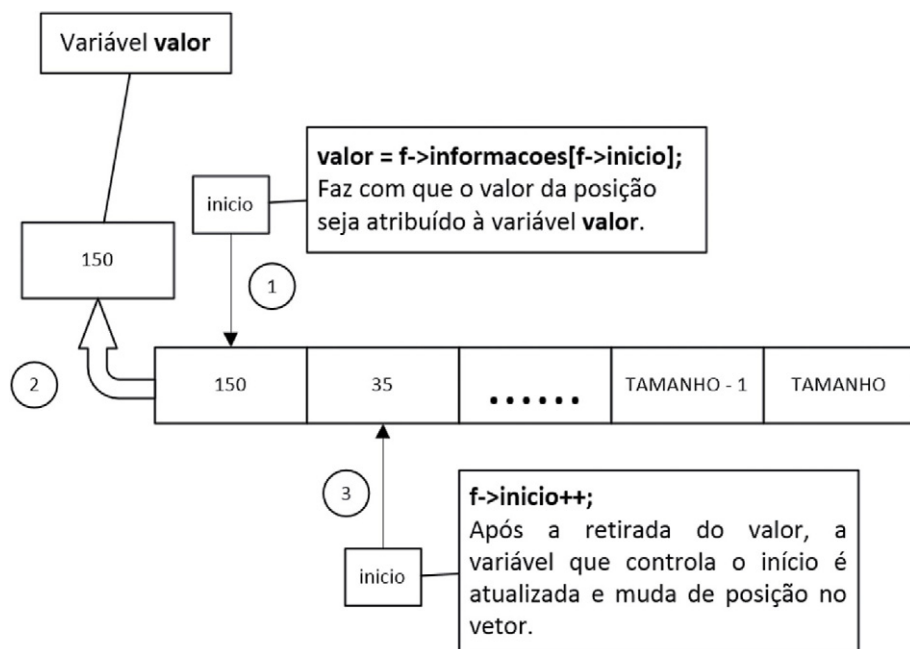


FIGURA 9.8: Sequência de retirada de um valor da fila.

Veja a sugestão desse código completo, com outros tratamentos e impressões de mensagens de erros encontrados:

Código 9.2

```
// Função para retirar/consumir um elemento da fila, se existir
int consumir(struct fila *f) // recebe o endereço da fila
{
    int valor; // cria uma variável local para receber o valor consumido
    if(f->inicio < TAMANHO) // verifica se não chegamos ao final da fila
    {
        if((f->inicio == 0) && (f->informações[f->inicio] == -1))
        {
            printf("Tentativa de consumir o primeiro elemento que ainda nao
existe. \n"); // Avisa que houve um problema
            valor = -1; // retorna um valor indicando problema
        }
        else
        {
            valor = f->informações[f->inicio]; // obtém o valor da posição e atribui na
// variável local
            f->inicio++; // atualiza a posição onde se encontra a posição inicial
            printf("<<< Consumido <<<: %d\n", valor); // imprimir na tela para controle
        }
    }
    else
    {
        printf("Tentativa de consumir uma informacao inesistente. \n\n"); // Avisa
// que houve um problema
        valor = -1; // retorna um valor indicando problema
    }
    return valor; // retornando valor consumido ou erro
}
```

Antes de entrar na função principal, o que você acha de fazermos uma função que nos auxilie a ver como a fila está? Estamos falando de fazer uma função de impressão da fila, onde ela pode nos mostrar quais informações (valores) estão na fila naquele momento. Acreditamos que isso possa facilitar nosso aprendizado.

Para imprimir o que está na fila, temos que percorrer a fila desde a posição inicial até a posição final, imprimindo os valores de cada posição

intermediária a elas (veja a Figura 9.9). Para fazer isso, precisamos de um comando de repetição; nesse caso, preferimos utilizar o comando *for*, mas você pode usar qualquer comando de repetição que conheça. Veja como esse comando de repetição ficou na nossa sugestão:

```
for(int i = f.inicio; i <= f.fim; i++){
```

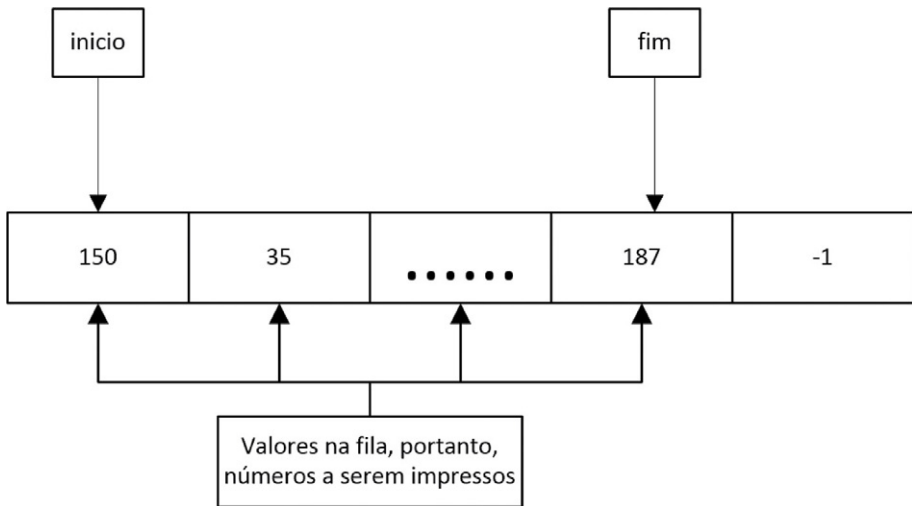


FIGURA 9.9: Cenário típico de impressão dos elementos que estão em uma fila, considerando que há elementos na fila.

Dentro desse comando, para melhorar nossa visualização nos exemplos, fizemos impressões diferenciadas para as posições inicial e final. Por isso, dentro do *loop* temos um comando *if*, *elseif* e um *else*. O primeiro *if* é para saber que estamos na posição inicial da fila; o comando *elseif* deve ser acionado apenas quando estivermos na posição final da fila; as outras posições são impressas no *else* final, portanto, esse trecho do código fica assim:

```
if(i == f.inicio) printf("inicio[%d]",f.informações[i]);  
else if(i == f.fim) printf(" fim[%d]",f.informações[i]);  
else printf(" %d",f.informações[i]);
```



Papo técnico:

Lembre-se de que, dentro de um comando condicional, como o `if`, caso tenhamos apenas uma linha de comando, não necessitamos dos “{” e “}” para delimitar o escopo. Isso também se aplica aos comandos de repetição, entre outros.

Antes de terminar essa função, é necessário identificar quando a fila está vazia e imprimir um aviso a esse respeito, pois nem sempre teremos uma impressão válida. Como podemos identificar que temos uma fila vazia? Pense um pouco sobre o posicionamento das posições inicial e final dentro do vetor. Qual é sua principal característica? Se você pensou que a posição inicial deve vir sempre antes da posição final, está no caminho certo. Nesse caso, verificaremos apenas se a posição inicial está à frente da posição final, por isso, nossa condicional ficou como apresentado a seguir. A [Figura 9.10](#) representa graficamente esse cenário.

```
if(f.inicio == (f.fim + 1))
```

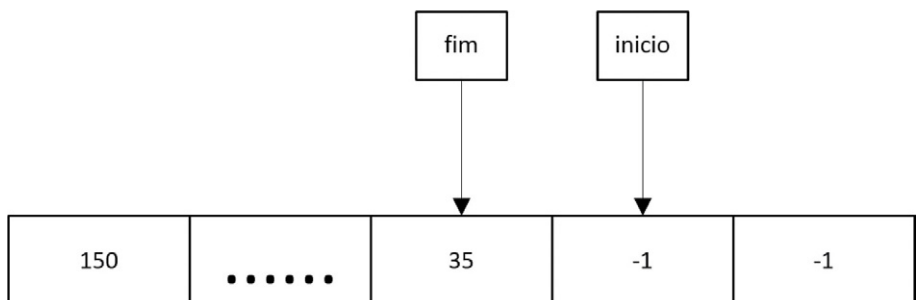


FIGURA 9.10: Com os ponteiros nessa posição, caracterizamos a fila vazia.

Apresento a seguir uma sugestão completa para essa função.

Código 9.3

```
// Função que imprime apenas os elementos existentes na fila
void imprimir(struct fila f)
{
    printf("\n\n----- Fila atual ----- \n");
    // Loop para imprimir na tela todos os elementos atuais da fila, por isso a variável de interação i
    // é inicializada com o valor da posição inicial da fila até a posição final.
    for(int i = f.inicio; i <= f.fim; i++)
    {
        if(i == f.inicio) printf("inicio[%d]", f.informações[i]);
        else if(i == f.fim) printf(" fim[%d]", f.informações[i]);
        else printf(" %d", f.informações[i]);
    }
    if(f.inicio == (f.fim + 1))
        printf("Fila vazia!");
    printf("\n\n----- \n\n");
}
```

Após definir as funções *inserção* na fila, consumo de uma informação da fila e também uma função que nos ajudará a visualizar nossa fila, gostaríamos de propor uma função principal que faça uso de todas elas. Mas não a use sem propósito exemplifique as formas de usá-la e faça alguns testes, ou seja, tente fazer algumas operações que não serão possíveis. Foi com esses dois objetivos que elaboramos essa função.

A primeira parte dessa função principal é a declaração de nossa fila e sua inicialização. Isso consiste em duas partes principais: a declaração de uma variável do tipo *struct* fila, que, no nosso caso, chamaremos simplesmente de *f*. Essa declaração fica assim:

```
struct fila f;
```



Papo técnico:

Lembre-se de que, ao se declarar uma variável, é alocada uma região de memória na *heap* suficientemente grande para conter todas as informações daquele tipo abstrato (nesse caso, não é um tipo primitivo). Portanto, ao declarar uma variável local *f*, já temos uma região de memória alocada para guardar nossa fila.

Após essa declaração, a segunda etapa é a inicialização dos elementos da nossa fila. Para a posição inicial, atribuiremos a posição zero, pois é a primeira posição em que colocaremos uma informação e, posteriormente, a obteremos. Já para a posição final, como temos um algoritmo que atualiza essa variável antes de atribuir o valor na posição que ela indica, temos de inicializá-la com -1, pois, após incrementada, pela primeira vez ela ficará com zero e será a posição inicial.

Já para o vetor que representa a fila, sugerimos que se faça uma inicialização com valores que teoricamente não serão usados. Nesse exemplo, assumimos que o valor -1 nunca será usado na fila como uma informação válida, por isso, o atribuímos em todas as posições.

Essas duas etapas estão contempladas nas linhas de código abaixo e representadas graficamente na [Figura 9.11](#):

```
struct fila f; // declaração da fila
f.inicio = 0;
f.fim = -1;
for(int i = 0; i < TAMANHO; i++)
    f.informações[i] = -1;
```

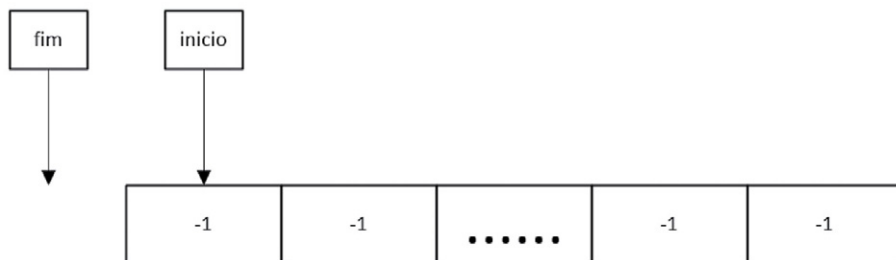


FIGURA 9.11: A fila, inicialmente, fica dessa maneira.

Depois da fase de declarações e inicializações, a fila está pronta para ser usada, portanto, já podemos manipulá-la. Como temos dois objetivos claros nesse exemplo – exemplificar o uso das funções de fila e testar alguns

comportamentos –, já iniciamos fazendo uso da função de consumir um valor sem sequer ter inserido algo. Nesse caso, como estamos usando a função *consumir* e ela retorna um valor consumido, necessitamos sempre atribuir o retorno dela em uma variável do tipo inteira, que é a nossa variável *valor*. Lembre-se de que essa função recebe o endereço de memória onde a fila está, pois necessita manipulá-la. Por isso passamos *&f* como parâmetro, ou seja, enviamos o endereço de memória onde a variável local *f* está alocada, e assim permitimos que a função faça o que for necessário com essa região de memória. Logo após o consumo de uma informação, fazemos a chamada para a função de impressão para verificar como se encontra a fila depois daquela passagem. O trecho de código dessa passagem é:

```
// Primeiro teste, o que acontece ao consumir algo de uma fila vazia?  
valor = consumir(&f);  
imprimir(f);
```

Será que você consegue identificar o que será mostrado na tela após essas duas linhas de código? Faça uma verificação nas funções envolvidas para ter uma ideia melhor e tente escrever essa saída em uma folha de seu caderno. Fazer esse teste é a melhor forma de entender o funcionamento da pilha.

Logo após esse primeiro teste, é feito uso da outra função que criamos, que é a de inserir um valor na fila. Essa função também recebe o endereço de memória da fila, pois, da mesma forma que a função de consumo, manipula as informações da fila e necessita acessar a memória diretamente. Como ela não retorna nenhuma informação, basta chamá-la passando o *&f* (endereço da variável *f*), ficando assim sua chamada:

```
inserir(&f, 10);
```

Apresentamos a seguir nossa sugestão completa para essa função principal. O que você acha de passar esse código linha a linha e escrever em uma folha de papel o que ele imprimirá na tela? Faça isso com calma e você verá que isso será bastante útil para o aprendizado total dos conceitos aqui apresentados. Boa sorte!

O código da função principal é:

Código 9.4

```
void main() {
    int valor = -1; // variavel que receberá a informação retirada da fila
    struct fila f; // declaração da fila
    f.inicio = 0; // inicializando a posição inicial como sendo a zero, ou seja, a posição
    f.fim = -1; // inicializando a posição final como sendo menos um, por que ao inserir um novo elemento essa
                // posição é incrementada e se tornará a posição zero

    // Loop de inicialização de todas as posições da fila
    for (int i = 0; i < TAMANHO; i++)
        f.informações[i] = -1;

    // Primeiro teste, o que acontece ao consumir algo de uma fila vazia?
    valor = consumir(&f);
    imprimir(f);

    //Inserindo 3 valores na fila
    inserir(&f, 10);
    inserir(&f, 20);
    inserir(&f, 30);
    imprimir(f);

    // Consumindo dois valores
    valor = consumir(&f);
    valor = consumir(&f);
    imprimir(f);

    // Por fim, inserindo mais dois valores e consumindo um
    inserir(&f, 40);
    inserir(&f, 50);
    valor = consumir(&f);
    imprimir(f);
}
```

Imaginando que você tenha passado esse código linha a linha, apresentando e escrevendo no papel o que era impresso, apresentamos na [Figura 9.12](#), para simples conferência, o que esse código resultou em nossos testes:

```
Tentativa de consumir o primeiro elemento que ainda nao existe.

----- Fila atual -----
Fila vazia!
-----

>>> Inserido 10>>>: 0
>>> Inserido 20>>>: 1
>>> Inserido 30>>>: 2

----- Fila atual -----
inicio[10], 20, fim[30]
-----

<<< Consumido <<<: 10
<<< Consumido <<<: 20

----- Fila atual -----
inicio[30],
-----

Estouro de fila
Estouro de fila
<<< Consumido <<<: 30

----- Fila atual -----
Fila vazia!
-----
```

FIGURA 9.12: Saída do programa exemplo no *prompt* de comando.

Nesse momento fechamos a discussão e exemplificação inicial a respeito de uma fila e sua implementação em vetor de forma contínua.

Para que possamos aprofundar um pouco o tema, gostaríamos de induzir você a ponderar a respeito de limitações no que foi apresentado até aqui. Quais foram as limitações que você percebeu nessa maneira de simular uma fila em programação? Pense alguns minutos a respeito. Temos certeza de que você reparará em vários fatores.

Muito bem. Provavelmente, o principal ponto que fica evidente é a limitação na quantidade de informações dentro da fila criada e manipulada. A utilização de vetor para essa função (controlar as posições da fila) traz esse problema intrinsecamente, mas podemos pensar em duas maneiras de tentar escapar dele.

A primeira visão que podemos ter é de que, apesar de o vetor estar limitando a quantidade de informações que serão vinculadas, nesse momento, temos um problema na forma como fizemos a implementação, pois, ao atualizar a posição de *inicio*, não usamos mais aquela posição do vetor. A [Figura 9.13](#) procura demonstrar e evidenciar as posições que estão desperdiçadas, o que pode ser encarado como problema, pois temos um pedaço de memória reservado e não usado – um desperdício, certo?

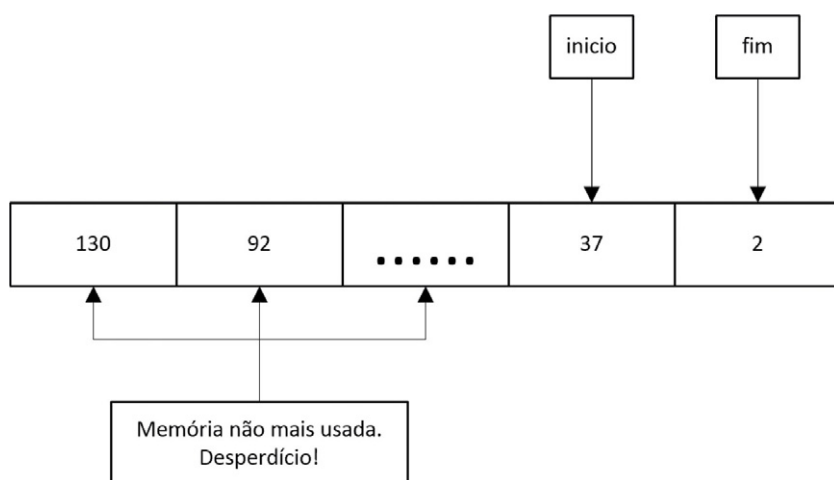


FIGURA 9.13: Posições já usadas desperdiçando espaço de memória.

Nós concordamos. Por isso, sugerimos a você que pense em uma implementação de vetor circular, ou seja, mesmo que o *final* da fila chegue ao final do vetor, se a posição *inicial* não estiver na posição zero do vetor, mesmo que o *final* da fila chegue ao final do vetor, se o *inicio* da fila não estiver nas posições iniciais do vetor, podemos deslocar o *final* da fila pelas posições em que o *inicio* da fila já tenha passado, fazendo com que o vetor fique circular. A [Figura 9.14](#) caracteriza essa circularidade dos ponteiros.

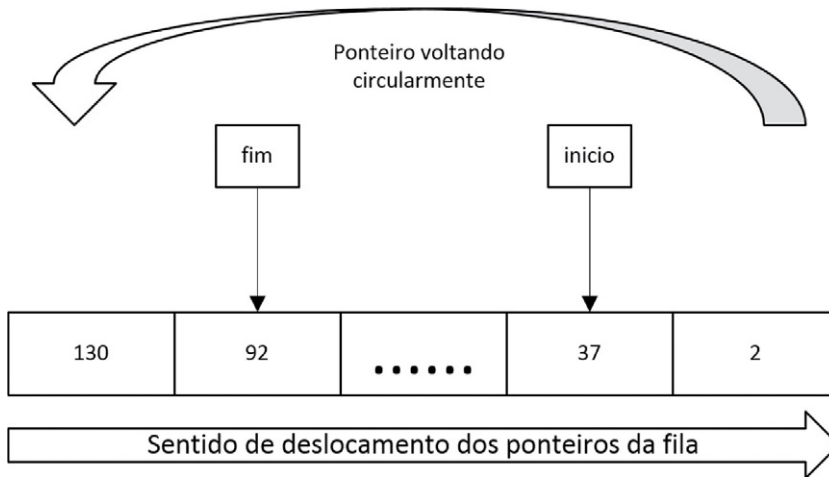


FIGURA 9.14: Vetor usado circularmente.

Na proposta apresentada, deve-se ter cuidado para que a posição *final* não ultrapasse a posição *inicial*, o que nos faz identificar outra limitação: o tamanho do vetor, nesse caso, é o tamanho máximo dessa fila.

Assim, para não haver limitação do tamanho da fila, uma ótima solução seria implementar essas mesmas funções levando em consideração o que foi aprendido e usado no capítulo 7 que fala sobre lista ligada. O que você acha de elaborar uma versão com essa nova possibilidade? Achemos que seria um exercício interessante para praticar e entender os conceitos aqui apresentados.



Vamos programar

Acabamos de ver a implementação de várias funções em linguagem C, mas, para que possamos ampliar nossos conhecimentos e horizontes, apresentamos as mesmas funções com a sintaxe nas linguagens JAVA e PYTHON. Aproveite para aprendê-las também.

Java

Código 9.1

```
private static void inserir (Queue<Integer> fila, int valor)
{
    try
    {
        fila.add(valor);
    }
    catch (Exception e)
    {
        System.out.println("Estouro de fila. Erro: "+e.toString());
    }
}
```

Código 9.2

```
private static int consumir (Queue<Integer> fila)
{
    try
    {
        return fila.poll();
    }
    catch (Exception e)
    {
        System.out.println("Tentativa de consumir uma informacao inesistente. Erro: "+e.toString());
    }
    return -1;
}
```

Código 9.3

```
private static void imprimir (Queue<Integer> fila)
{
    System.out.println(fila);
}
```

Código 9.4

```
public static void main (String[] args)
{
    Queue<Integer> fila = new LinkedList<Integer>();
    int valor;
    // Primeiro teste, o que acontece ao consumir algo de uma fila vazia?
    valor = consumir(fila);
    System.out.println(fila);

    //Inserindo 3 valores na fila
    inserir(fila, 10);
    inserir(fila, 20);
    inserir(fila, 30);
    imprimir(fila);

    // Consumindo dois valores
    valor = consumir(fila);
    valor = consumir(fila);
    imprimir(fila);

    // Por fim, inserindo mais dois valores e consumindo um
    inserir(fila, 40);
    inserir(fila, 50);
    valor = consumir(fila);
    imprimir(fila);
}
```

Python

Código 9.1

```
def inserir(self, valor):
    if self.fim + 1 < TAMANHO: #verifica se nao chegamos ao final da fila
        self.fim+=1 #atualiza a posicao final para inserir um novo valor
        self.informacoes[self.fim] = valor #insere o valor na posicao
        print "Inserido : %d" % (valor) #imprimi na tela para controle
    else:
        print "Estouro de fila" #avisa que houve estouro de fila
```

Código 9.2

```
#Funcao para retirar/consumir um elemento da fila, se existir
def consumir(self):
    valor=0 #cria uma variavel local para receber o valor consumido
    if self.inicio < TAMANHO: #verifica se não chegamos ao final da fila
        if self.inicio == 0 and self.informacoes[self.inicio] == -1:
            print "Tentativa de consumir o primeiro elemento que ainda nao existe"
            valor = -1 #Avisa que houve um problema
        else:
            valor = self.informacoes[self.inicio] #obtem o valor da posicao e atribui na variavel local
            self.inicio+=1 #atualiza a posicao onde se encontra a posicao inicial
            print "Consumido %d" % (valor) #imprimi na tela para controle
    else:
        print "Tentativa de consumir uma informacao inexistente"
        valor = -1 #retorna um valor indicando problema
    return valor #retornando valor consumido ou erro
```

Código 9.3

```
#Funcao que imprime apenas os elementos existentes na fila
def imprimir(self):
    print "Fila Atual"
    i = 0
    while i <= self.fim: # Loop para imprimir na tela todos os elementos atuais da fila, por isso a variavel de
        # interação i é inicializada com o valor da posição inicial da fila até a posição final
        if i == self.inicio:
            print "inicio[%d]" % (self.informacoes[i])
        elif i == self.fim:
            print "fim[%d]" % (self.informacoes[i])
        else:
            print self.informacoes[i]
        i += 1
    if self.inicio == self.fim + 1:
        print "fila vazia"
    print "fim"
```

Código 9.4

```
#DEFINICAO DA ESTRUTURA
class Fila:
    def __init__(self):
        self.informacoes = [-1]*TAMANHO
        self.inicio = 0
        self.fim = -1

    .
    .
    .
    #Funções 9.1, 9.2 e 9.3 seriam colocadas aqui.
    .
    .
    .

    fila.consumir()
    fila.imprimir()

    fila.inserir(num)
    fila.inserir(num)
    fila.inserir(num)
    fila.imprimir()

    fila.consumir()
    fila.consumir()
    fila.imprimir()

    fila.inserir(num)
    fila.inserir(num)
    fila.imprimir()
```



Para fixar

1. Implemente seu próprio programa usando qualquer uma das propostas apresentadas neste capítulo.
2. Em seguida, crie uma rotina para a fila do restaurante na praça de alimentação, mas leve em consideração que ele tem apenas um atendente no caixa. Depois de ser atendido no caixa, o cliente passa a outra fila, para receber seu pedido. Nessa segunda fila, temos três atendentes.



Para saber mais

Alguns de vocês podem questionar a validade de uma estrutura de dados como a fila, mas ela é muito mais utilizada do que se pode imaginar.

Geralmente, em *sistemas operacionais* ou mesmo em *redes de computadores*, os recursos compartilhados têm políticas de acesso baseadas em filas, ou seja, os primeiros a chegar são os primeiros a ter os recursos disponibilizados para seu uso. Sem dúvida, existem recursos com várias filas, cada uma delas com prioridades diferentes. Mesmo assim, o compartilhamento do recurso está baseado em filas.

Um exemplo simples e corriqueiro dessa utilização é o *servidor de impressão* de seu laboratório de informática. Vários alunos podem mandar seus arquivos para serem impressos ao mesmo tempo, mas nem por isso os textos são impressos de forma misturada. Cada um dos arquivos é impresso de uma vez, sem ter as partes misturadas. Isso se deve ao enfileiramento dos arquivos que chegam ao servidor e são consumidos pela impressora.

Outro recurso, hoje muito importante, que se utiliza do enfileiramento das informações que chegam é o processador, em que o sistema operacional deve gerenciar uma boa quantidade de processos aptos a serem processados. Dentre as várias formas, a fila é uma das mais usadas.

Em redes de computadores, as filas são muito usadas em algoritmos de comunicação de dados, principalmente para receber, processar e enviar pacotes que trafegam pelos equipamentos de núcleo de rede. A fila de entrada dos servidores é extremamente explorada para fazer ataques a esses recursos. Como a fila, por melhor implementada que seja, esbarra na limitação de memória do servidor, os atacantes sempre tentam explorar seu congestionamento total para fazer com que o servidor pare de responder e inicie o processo de negar as novas requisições feitas, levando o cliente a

obter a informação de que o servidor está indisponível naquele momento, ou mesmo inutilizando um importante nó do núcleo da rede, provavelmente congestionando em cascata os demais nós da rede.

Por este último exemplo, podemos perceber que as filas são bastante importantes. Imagine se nossas redes fossem vulneráveis dessa forma e tivéssemos um colapso total na internet, a ponto de não podermos acessar momentaneamente nossas redes sociais. Como seria isso?

Existe uma interessante discussão sobre *fila de prioridade* no Capítulo 4 do livro *Estrutura de dados usando C* (Tenenbaum, 2004).



Navegar é preciso

Para se aprofundar no tema, saber um pouco sobre a famosa “teoria das filas” é muito importante. Para isso, acesse a discussão presente em:

http://www.eventhelix.com/RealtimeMantra/CongestionControl/queueing_theory.htm#.UhAnJ3-YdD0.

Os simuladores presentes em <http://queueing-systems.ens-lyon.fr/> dão exemplos de vários tipos de fila.

Exercícios

1. Tente simular uma fila de banco, nesse caso, apenas uma fila de clientes e dois caixas de atendimento.
2. Acrescente à simulação anterior uma fila prioritária.
3. Adicione também um caixa de atendimento. Ele deve ser o único caixa a atender à fila prioritária.
4. Por fim, acrescente as impressões de tempo de espera de cada cliente. Considere que chega um cliente a cada segundo, e que cada um dos caixas atende a um cliente a cada dois minutos.
5. Crie uma estrutura cliente em que se tenha o número de chegada dele e o tempo que ele estima usar no caixa (o cliente deve saber o tempo que vai demorar no caixa). Simule um atendimento com quatro caixas e uma única fila, que receba com frequência clientes com diferentes tempos de atendimento (você pode escolher como fará essa atribuição, porém, ela tem que ser aleatória).
6. Elabore um programa que simule um caixa que demora sempre o mesmo tempo para atender a um cliente. O problema é que a frequência de chegada de novos clientes pode variar e fazer com que a fila cresça. Implemente essa fila com um limite de 10 clientes e que tenha vetor circular.

Glossário

FIFO: *First In, First Out* – o primeiro que entra é o primeiro a sair.

FILO: *First In, Last Out* – o primeiro que entra é o último a sair.

Referências bibliográficas

PIVA, D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Campus, 2012.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.



QUE VEM DEPOIS

Você já conhece várias maneiras de fazer busca, ordenação e listas ligadas, além das filas. No Capítulo 10 será a vez de aprender um pouco sobre uma nova estrutura de dados chamada *pilha*.

Pilhas

“Os últimos serão os primeiros.”

BÍBLIA (MATEUS 20:1-16 / MATEUS 19:23-30 / MARCOS 10:23-31 / LUCAS 13:23-30)

Todos nós já ouvimos essa frase, mas o que muitos não sabem é que nela está contida a ideia básica da estrutura de dados que veremos neste capítulo. Vamos perceber que as últimas informações guardadas serão as primeiras a serem utilizadas e, melhor, que essa característica nos ajudará a resolver alguns problemas computacionais.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o funcionamento das pilhas;
- identificar situações e formas de utilizar pilhas;
- entender como os recursos computacionais utilizam-se das filas para solucionar problemas.



Para começar

A pilha é uma estrutura de dados muito frequente em nosso dia a dia, por isso seu entendimento é fácil e faz com que, de certa forma, sua implementação seja bastante lógica e intuitiva. Afinal, depois de entender um algoritmo em detalhes, a implementação é “fichinha”, não é mesmo?



Para um bom entendimento, vamos começar ilustrando um cenário que pode ocorrer em algum momento de nossas vidas, principalmente na daqueles que já se mudaram de residência pelo menos uma vez. Alguns devem estar pensando: “Nossa, nunca me mudei! E agora?”. Calma! Aqueles que nunca passaram por isso podem ficar tranquilos, pois a situação é facilmente imaginável. Certamente, algo bem parecido já ocorreu quando você decidiu arrumar algum cômodo do lugar onde mora.

Para entender como funciona a dinâmica de uma pilha, imagine a mudança de uma residência para outra. Considere que você esteja de mudança e que seus inúmeros livros sairão de uma estante acanhada no canto de seu quarto e irão para um lugar especial em sua nova biblioteca pessoal – afinal, bons livros, como este, devem ter um local de destaque em sua nova residência.

Imagine que, dias antes da mudança, a empresa contratada para transportar seus pertences lhe forneceu várias caixas de papelão, em três tamanhos diferentes. Os diferentes tamanhos visam a acomodar adequadamente

as roupas, os utensílios e, entre outras coisas, seus estimados livros. Ao analisar os tamanhos disponíveis, você constata que apenas um deles serve para esse transporte e mais: o tamanho de caixa disponível permite apenas que os livros sejam acondicionados uns sobre os outros, ou seja, que não é possível acomodar um livro ao lado de outros.



Papo técnico

Nessa analogia, a caixa representa um espaço de memória, e os livros são as informações que desejamos guardar.

Nesse cenário, o que fica claro para nós é que, dentro da caixa, um livro ficará em cima do outro, independentemente de seu tamanho, certo? Obviamente, esse é um cenário hipotético, mas não está muito fora da realidade.

Como ocorre esse acondicionamento dos livros nas caixas fornecidas? Quais passos você seguiria? Sem dúvida, o primeiro passo seria escolher uma das prateleiras da estante e decidir qual sequência de livros deslocar para a caixa. O segundo seria o processo de retirar, um a um, os livros da estante e colocá-los na caixa. O primeiro livro será colocado no fundo da caixa. Depois dele, os demais serão acomodados uns em cima dos outros, ou seja, no topo da pilha de livros que está sendo formada.

O processo de guardar livros nas caixas é hipoteticamente fixo, ou seja, um livro é selecionado na estante e sempre acomodado no topo da pilha de livros formada dentro da caixa. Dentro da caixa, sempre que um livro entra, vira o topo atual e permanece nesse posto até que outro livro seja acomodado em cima dele. Esse processo ocorrerá enquanto a caixa tiver espaço para acomodar mais livros. Quando a pilha de livros ocupar o espaço total da caixa, é melhor fechá-la e deixá-la pronta para a mudança.

Logo após a mudança, já em sua maravilhosa e novíssima biblioteca, as caixas com seus estimados livros estão disponíveis, aguardando para serem esvaziadas e, assim, deixar sua biblioteca recheada com seus exemplares.

Depois de escolher uma das caixas e abri-la, você começa a retirar um livro após o outro – antes de guardá-los nas novíssimas estantes, é necessário limpá-los. Por isso, o procedimento de desmontar essas caixas é retirar do topo da pilha de livros um volume, limpá-lo e acomodá-lo no novo local. Esse procedimento será feito para cada um dos livros que estão dentro da caixa. Portanto, um livro é retirado do topo da pilha, limpo e, por fim, acomodado na prateleira da estante.

**Atenção**

Repare que, na pilha de livros que se formou dentro da caixa, o último livro colocado foi o primeiro a ser retirado.

**Conhecendo a teoria para programar**

No exemplo hipotético, vimos que os livros foram acomodados no topo da pilha que existia dentro da caixa e, quando estavam sendo acomodados na nova biblioteca, foram retirados do topo da pilha de que faziam parte.

Depois de entender essa dinâmica toda, devemos pensar em como implementá-la. Mas, antes disso, gostaríamos de analisar algumas partes. Em nosso exemplo, a caixa era o artefato que fazia nossos livros serem armazenados. Para nós, em programação, a caixa faz o papel de memória, onde armazenamos informações (no nosso exemplo, os livros). Portanto, no exemplo temos caixas armazenando livros, ao passo que em programação temos memória armazenando quaisquer informações que desejarmos.

A informação que será armazenada na memória pode ser de qualquer tipo primitivo, mas também é aceito qualquer tipo abstrato desenvolvido em seu programa.

Antes de iniciar as implementações de funções importantes, gostaríamos de discutir o tamanho de nossa pilha. Em nossos exemplos usaremos um vetor como alocação de memória para trabalhar a pilha. Um vetor, ao ser declarado, deve ter tamanho definido, que será o tamanho máximo da

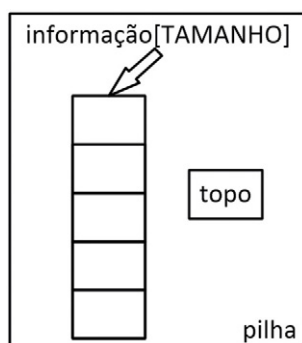


FIGURA 10.1: Pilha representada graficamente.

nossa pilha, ou seja, a quantidade máxima de elementos que caberão ali. A [Figura 10.1](#) mostra como ficará a nossa pilha.

Nesse caso, faremos uso da declaração de uma constante que conterà esse valor. O código ficará assim:

```
#define TAMANHO 4
```

Além do tamanho, o vetor também deve ser um tipo. No nosso caso, nós o criaremos com um tipo primitivo, inteiro, mas nossa pilha será definida em um tipo abstrato, contendo uma variável que controlará a posição do topo da pilha e o vetor de inteiros que servirá de pilha em nossos exemplos. O código apresentado a seguir demonstra a definição dessa estrutura:

```
struct pilha
{
    int topo;
    int informacao[TAMANHO];
};
```

A variável que controla o topo da pilha sempre indicará a posição dessa pilha que receberá a informação seguinte. Dessa forma, se iniciarmos seu valor com 0 (zero), obviamente ela nunca será menor do que isso, porém, na outra extremidade, quando a pilha estiver cheia, terá valor igual ao tamanho máximo. Calma. Como assim terá o valor igual ao tamanho máximo? (No nosso caso, 4.) Sim, isso mesmo. Lembre-se de que se trata de um vetor de quatro posições, portanto, as posições existentes iniciam em 0 (zero) e terminam em 3 (três). Agora, fica bem claro que, quando a variável controladora do topo estiver com valor igual ao tamanho máximo, isso significa que nossa pilha está cheia e que não podemos receber mais valores.

Levando em conta que não é possível apenas inserir valores, isto é, que também é possível retirá-los, é importante salientar que essa variável indica uma posição que não tem um valor a ser considerado. Portanto, para obter a informação que se encontra no topo, basta atualizar essa variável, decrementando seu valor para chegar a uma posição com valor e, aí sim, retirá-lo.

Na [Figura 10.2](#), vemos como exemplo a evolução de uma pilha. Representamos uma pilha de quatro posições. A [Figura 10.2\(a\)](#) demonstra como essa

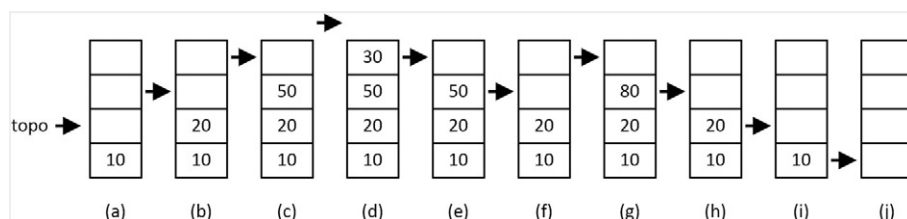


FIGURA 10.2: Evolução do topo da pilha durante ações de *push* e *pop* na pilha.

pilha ficaria com uma informação armazenada. Repare que temos o valor 10 (dez) na primeira posição da pilha (index zero do vetor) e o ponteiro para o topo encontra-se em uma posição sem informação (index número um do vetor), pois o próximo elemento, se existir, entrará nessa posição. As Figuras 10.2(b, c, d) ilustram a evolução da pilha ao receber três informações em sequência, que a deixam cheia. Na posição caracterizada na Figura 10.2(d), temos a informação 30 completando a pilha, e nossa variável de indicação de topo está apontando para uma posição que não existe.

Fazer a variável de topo armazenar uma posição que não existe é errado? O que você acha? Lembre-se de que, se você usar de fato essa posição do vetor, ela não existirá, portanto, você acessará uma região de memória inválida para seu vetor. De outro lado, apenas ter a informação guardada para indicar que a pilha está cheia, sem acessar nenhuma posição, é possível e extremamente válido.

Bem, já vimos uma característica importante de nossa pilha, ou seja, quando ela está cheia – Figura 10.2(d). Agora, analisaremos as Figuras 10.2(e, f). Repare que o topo está descendo – na verdade, decrementando – e alguns valores estão sendo retirados do topo. Essas duas figuras nos mostram que, para retirar uma informação do topo da pilha, devemos primeiramente decrementar a variável indicativa de posição de topo, guardar a informação que está naquela posição e, depois, apagá-la, colocando um valor inválido ou zerando.

A Figura 10.2(g) mostra o caso de termos um novo elemento sendo colocado na pilha, e as Figuras 10.2(h) e (i) mostram retiradas simples de informações da pilha. Já a Figura 10.2(j) demonstra a última retirada possível dessa sequência, pois deixa a pilha vazia. Repare que a posição de topo está na posição zero do vetor, o que caracteriza essa situação.

Espero que esse exemplo tenha deixado bastante claro os principais passos que devemos seguir para inserir uma informação em uma pilha ou retirá-la dela.

A partir deste momento, não falaremos mais em *inserir* e *retirar*; falaremos de como essas operações são conhecidas no mundo técnico, ou seja, para dizer que inserimos algo na pilha, falaremos em *push*; e para dizer que retiramos algo da pilha, usaremos o termo *pop*.

Função *push*: inserindo novo elemento na pilha

Continuando nosso código, gostaríamos de discutir a função *push*. Nela, temos a necessidade de inserir nova informação dentro da nossa pilha. Para tal, devemos receber a pilha propriamente dita, pois, se formos adicionar uma informação, devemos saber onde, concorda? Além disso, precisamos saber o que será adicionado.

Imagine que seu professor solicite a você que se levante e escreva algo. A primeira coisa que você perguntará é: “Onde vou escrever?” Ao receber

a resposta dele, de que será na lousa, você logo formula mentalmente a segunda pergunta: “E o que eu vou escrever ali?”. Suponha que o professor solicite que você escreva quantos títulos de campeonato brasileiro de futebol seu time de coração já conquistou. Ótimo, já temos local para escrever e também conteúdo. Agora ficou fácil, não é? Basta encaminhar-se até o local indicado e escrever o que lhe foi solicitado.

Pois bem, com a função *push* ocorre a mesma coisa. Ela receberá uma pilha já definida por quem está chamando essa função (a lousa do nosso exemplo) e também a informação que será inserida no topo (quantidade de títulos conquistados por seu time de coração no campeonato brasileiro de futebol). Caso seja possível, o cabeçalho dessa função fica assim:

```
void push(struct pilha *p, int info)
```

Após receber a pilha e a informação que deve ser colocada nela, é necessário verificar se essa pilha não está cheia. Se estiver, não há o que fazer, a não ser dar uma mensagem informativa; caso contrário, isto é, se a pilha puder receber essa informação, devemos realizar o procedimento de inserção.

Para verificar, cria-se um simples código condicional questionando se o topo é menor que o tamanho máximo da pilha. assim, o condicional ficaria dessa forma:

```
if (p->topo < TAM)
```

Sendo o topo menor que o tamanho máximo, devemos fazer a inserção da informação dentro da pilha, seguindo os passos já explicados e exemplificados na [Figura 10.3](#), ou seja, colocar o valor na posição sinalizada pelo topo (ver [Figura 10.3\(a\)](#)) e atualizar o topo (ver [Figura 10.3\(b\)](#)), incrementando-o e deixando-o pronto para a próxima inserção de valores.

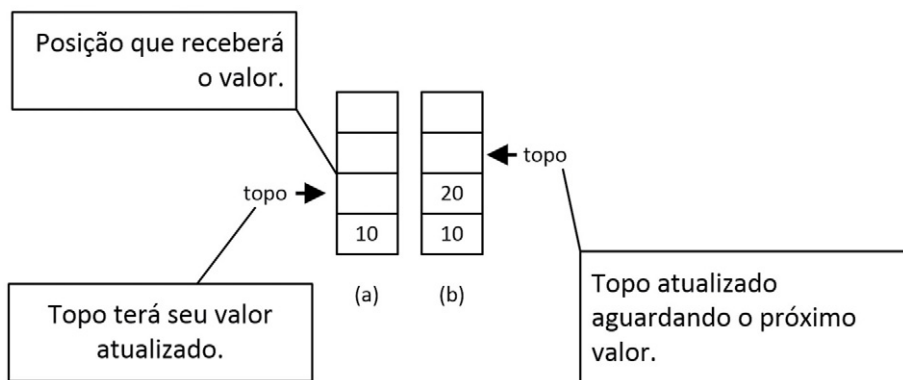


FIGURA 10.3: Ilustração dos passos para inserção de um novo elemento na pilha.

As duas linhas de código que fazem esse procedimento são as seguintes:

```
p->informacao[p->topo] = info;  
p->topo++;
```

Se o topo indicar que a pilha está cheia, resta apenas enviar uma mensagem na tela; assim, o código ficaria:

```
else  
    printf("Pilha cheia, impossivel INSERIR elemento!\n\n");
```

A função completa do *push* ficaria assim:

Código 10.1

```
void push(struct pilha *p, int info)  
{  
    //Verifica se há espaço na pilha para armazenar nova informação  
    if(p->topo < TAM)  
    {  
        printf(">>>>> INSERIR >>> : %d\n", info);  
        p->informacao[p->topo] = info; //a variável "topo" aponta para a posição onde a nova  
        // informação deve ser inserida, portanto, devemos fazer a atribuição.  
        p->topo++; //atualiza a variável que controla o "topo" para que se possa receber a próxima informação  
    }  
    else  
        printf("Pilha cheia, impossivel INSERIR elemento!\n\n");  
}
```

Função pop: retirando elemento na pilha

Agora é a vez de discutirmos e apresentarmos a função *pop*. Ela exerce a funcionalidade de retirar uma informação do topo da pilha, como já discutimos. Você se lembra disso, né? Certamente que sim.

A função *pop* deve retornar uma informação retirada do topo da pilha. Por isso, ela tem o tipo primitivo *int* no seu cabeçalho, indicando que retornará um número inteiro. Já em seus parâmetros, ele recebe

um endereço de pilha, pois também fará manipulações nela (mudando o topo e retirando um elemento). Considerando isso, nosso cabeçalho ficará assim:

```
int pop(struct pilha *p)
```

Nessa função, ao contrário da função *push*, precisamos verificar se estão solicitando a remoção de um elemento de uma pilha vazia. Para fazer isso, basta verificar se o topo não está na posição zero da pilha. Essa verificação é feita por uma simples condicional, como:

```
if (p->topo > 0)
```

Se houver elemento(s) a ser(em) retirado(s), deve-se realizar o procedimento de retirada, ou seja, atualizar o topo decrementando-o, guardando o valor que está na posição indicada em uma variável, atribuindo um valor que não representa nada dentro da pilha e retornando dessa função com o valor que foi retirado do topo. A [Figura 10.4](#) mostra os passos descritos. Na [Figura 10.4\(a\)](#) temos a pilha com o elemento que será retirado do topo (número 20), o posicionamento do ponteiro de

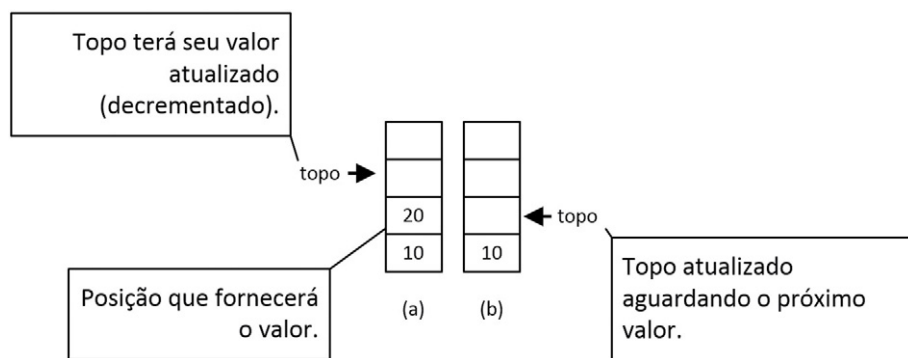


FIGURA 10.4: Passos para a remoção de um elemento da pilha.

topo; a [Figura 10.4\(b\)](#) mostra a situação dessa pilha após a retirada do elemento.

```
p->topo--;
valor = p->informacao[p->topo];
p->informacao[p->topo] = -1;
return valor;
```

Se a condicional indicar que não há mais elementos a serem retirados, imprimiremos uma mensagem ao usuário avisando-o disso.

Assim, o código ficaria:

```
else
    printf("Pilha vazia, impossivel RETIRAR elemento!\n\n");
```

Caso o código chegue até esse ponto, isso significa que não houve retorno de um elemento válido a quem chamou essa função, portanto, é melhor retornar um valor inválido.

Após essas explicações dos trechos de códigos, apresentamos a função proposta completa:

Código 10.2

```
int pop(struct pilha *p)
{
    int valor; //variável que guardará a informação que será retirada da pilha
    //Verifica se existe alguma informação a ser retirada na pilha
    if(p->topo > 0)
    {
        p->topo--; //atualiza o topo para a posição onde temos a última informação inserida
        valor = p->informacao[p->topo]; //obtem a informação
        p->informacao[p->topo] = -1; //atualiza a posição com a informação que não representa nada
        printf("<<< RETIRAR <<<<< : %d\n", valor);
        return valor; //retorna o valor para quem chamou a função pop
    }
    else
        printf("Pilha vazia, impossivel RETIRAR elemento!\n\n");
    return -1;
}
```

Função principal: usando *push* e *pop*

Antes de chegar à função principal, para explicar melhor nossos testes, gostaríamos de apresentar uma função que visa única e exclusivamente a mostrar, no terminal de execução desse programa, os elementos que estão dentro dessa pilha. É preciso deixar bem claro que essa função não existe quando se trata de pilha, pois dentro das regras citadas o único elemento visível dessa estrutura de dados é seu elemento caracterizado como topo.

Para essa função, usaremos basicamente a liberdade que nos é dada pelo vetor para sair da posição zero e, avançando posição a posição, chegar até uma delas imprimindo seus elementos. Nesse caso, deixaremos a posição zero visando chegar àquela que está sinalizada como a posição topo.

A função de impressão sugerida é:

Código 10.3

```
int i = 0, valor_retirado;  
struct pilha pi; //declaração da variável que guardará a nossa pilha
```

Agora, vamos para a função principal, em que utilizaremos todas as funções apresentadas até aqui. É nessa função que vamos criar nossa pilha e também a variável auxiliar que guardará a informação retirada dessa pilha, quando for solicitada. Para tais declarações, sugiro as seguintes linhas de código:

```
void imprimir(struct pilha p)  
{  
    printf("***** PILHA *****\n\n");  
    //repetição entre as posições que contém informações para que sejam impressas na tela  
    for(int i = 0; i < p.topo; i++)  
        printf(" %d", p.informacao[i]);  
    printf("\n\n");  
    printf("*****\n\n");  
}
```

Após a criação da pilha e das variáveis auxiliares, é necessário inicializá-la. Certo, mas inicializar a pilha em que sentido? Você se lembra de que tínhamos uma variável interna que controlava a posição do vetor que representava o topo da pilha? Pois então. Essa variável deve ser inicializada com o valor zero. Você lembra por que justamente o valor zero? É porque essa posição receberá a primeira informação a entrar na pilha.

Aliás, e o vetor que representa a pilha, não precisa ser inicializado? Na verdade, isso não é uma regra, mas é uma boa prática. Nesse caso, um valor que representará um valor inválido dentro desse exemplo é -1, portanto, ele fará parte de todas as posições inicialmente.

Para essas inicializações necessárias à nossa pilha, sugerimos as seguintes linhas de código:

```
pi.topo = 0;
for(i = 0; i < TAM; i++)
    pi.informacao[i] = -1;
```

Após essas partes iniciais e importantes da nossa função principal, sugerimos que se faça uma sequência de inserções na pilha, chegando ao ponto de inserir um elemento a mais que ela suporte para testar o comportamento da função *push*. Para facilitar o entendimento do nosso teste, propomos fazer a sequência de inserções na pilha sugeridas na [Figura 10.2](#). O código ficaria assim:

```
//sequência de inserções de valores na pilha
push(&pi, 10);
imprimir(pi);
push(&pi, 20);
imprimir(pi);
push(&pi, 50);
imprimir(pi);
push(&pi, 30);
imprimir(pi);
//esta inserção é um teste, pois passa do tamanho da pilha
push(&pi, 40);
imprimir(pi);
```

Para a função *push*, temos que passar como primeiro parâmetro o endereço de memória onde se encontra a nossa pilha, pois faremos a manipulação de seu conteúdo e gostaríamos que essa pilha fosse atualizada para todos os que a usarão. Por isso, o primeiro parâmetro é o *&pi*, que envia para a função *push* o endereço da pilha declarada e chamada, nessa função principal, simplesmente de *pi*.

Agora que fizemos as inserções necessárias, sugiro passar para uma sequência que retira da pilha as informações nela contidas. Assim como a função *push*, a função *pop* manipula a mesma pilha, portanto, deve receber seu endereço. Aproveitando a ideia já utilizada na função *push*, propomos também uma retirada a mais da pilha para testar o comportamento da função *pop*. Repare que essa função retorna um valor toda vez que é chamada, por isso colocamos a variável *valor_retirado* para receber a atribuição do valor que retorna dessa chamada de função. Nesse caso, tal variável não é utilizada para nada, mas poderia ser empregada por alguma lógica específica que usasse as pilhas como parte de sua possível solução.

A sequência que sugiro é:

```
//sequência de retiradas dos valores da pilha
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
valor_retirado = pop(&pi);
imprimir(pi);
//este valor não deve existir
valor_retirado = pop(&pi);
imprimir(pi);
```

Portanto, o código completo dessa função principal é:

Código 10.4

```
int _tmain(int argc, _TCHAR* argv[])
{
    int i = 0, valor_retirado;
    struct pilha pi; //declaração da variável que guardará a nossa pilha

    //Inicializando a pilha recém criada.
    //A variável "topo" guarda a posição que receberá a informação e depois será atualizada novamente
    pi.topo = 0;
    //Inicializando as posições do vetor que funcionará como pilha
    for(i = 0; i < TAM; i++)
        pi.informacao[i] = -1;

    //sequência de inserções de valores na pilha
    push(&pi, 10);
    imprimir(pi);
    push(&pi, 20);
    imprimir(pi);
    push(&pi, 50);
    imprimir(pi);
    push(&pi, 30);
    imprimir(pi);
    //esta inserção é um teste, pois passa do tamanho da pilha
    push(&pi, 40);
    imprimir(pi);

    //sequência de retiradas dos valores da pilha
    valor_retirado = pop(&pi);
    imprimir(pi);
    valor_retirado = pop(&pi);
    imprimir(pi);
    valor_retirado = pop(&pi);
    imprimir(pi);
    valor_retirado = pop(&pi);
    imprimir(pi);
    //este valor não deve existir
    valor_retirado = pop(&pi);
    imprimir(pi);

    return 0;
}
```

Apresentaremos, a seguir, o resultado da execução desse código. Observe que as funções *push* e a *pop* se comportaram como esperado, pois emitiram frases informando que não foi possível inserir e retirar elementos, e, em nenhum dos casos, o programa perdeu dados, muito menos teve um comportamento inadequado. Observe a saída do programa proposto (ver Figura 10.5)

O trecho de código que exerce esses passos é:

```

>>>> INSERIR >>> : 10
***** PILHA *****
10
*****
>>>> INSERIR >>> : 20
***** PILHA *****
10 20
*****
>>>> INSERIR >>> : 50
***** PILHA *****
10 20 50
*****
>>>> INSERIR >>> : 30
***** PILHA *****
10 20 50 30
*****
Pilha cheia, impossivel INSERIR elemento!
***** PILHA *****
10 20 50 30
*****
<< RETIRAR <<<< : 30
***** PILHA *****
10 20 50
*****
<< RETIRAR <<<< : 50
***** PILHA *****
10 20
*****
<< RETIRAR <<<< : 20
***** PILHA *****
10
*****
<< RETIRAR <<<< : 10
***** PILHA *****
*****
Pilha vazia, impossivel RETIRAR elemento!
***** PILHA *****
*****

```

FIGURA 10.5: Resultado da execução do código teste.

Pilha na prática: rádio? Não, expressão matemática mesmo!

As pilhas são usadas como parte da solução de um relevante tópico em ciência da computação, que é a análise de expressões matemáticas, principalmente ao pensar que essa análise pode fazer parte de um compilador de linguagem, aumentando ainda mais sua importância.

As expressões são definidas de três maneiras: *infixo*, *prefixo* e *posfixo*. Os prefixos *in*, *pre* e *pos* referem-se, única e exclusivamente, ao posicionamento dos operadores matemáticos em relação a seus dois operandos. Veja os exemplos a seguir:

- infix: $X + Y + Z$
- posfix: $X Y Z + +$
- prefix: $+ + X Y Z$

A expressão, da forma como estamos acostumados a ler e usar, é representada pela forma infix, como $X + Y$. Como descrito no parágrafo anterior, os prefixos referem-se ao *posicionamento dos operadores em relação aos operandos*, portanto, se essa mesma expressão do exemplo fosse escrita na forma *prefixa*, a expressão ficaria $+ X Y$; de outro lado, na forma *posfixa*, a expressão ficaria $X Y +$.

Até aqui tudo muito simples, certo? Pois é. Mas como em computação nada é tão simples assim, vamos adicionar algo para pensar em como resolver. Outro exemplo no formato *infixo* seria $X + Y * Z$. Nós temos a nítida impressão de que devemos resolver primeiro $Y * Z$ para depois somar o resultado ao X , mas como fazer o computador saber o mesmo que nós? Podemos dizer, então, que sabemos que as operações de multiplicação e divisão têm *precedência* sobre as operações de soma e subtração. Nesse exemplo, a forma *posfixa* resultaria em $X Y Z * +$, pois, dessa forma, o Y e o Z seriam multiplicados e o resultado seria somado com X , obtendo, assim, a resposta pretendida.

Conversão de infix para posfixo

Para que você possa entender como essa transformação acontece e o que a pilha tem a ver com tudo isso, apresentamos um algoritmo para fazer exatamente essa conversão, usando a estrutura de dados que acabamos de ver, ou seja, a pilha.

Antes de iniciar, devemos supor que exista uma função chamada $\text{precd}(\text{op1}, \text{op2})$, que retorna um *tipo booleano*, sendo *TRUE* se op1 tiver precedência sobre op2 , e *FALSE* se ocorrer o contrário. Por exemplo, $\text{precd}(*, +)$ retornará *TRUE*, e $\text{precd}(+, *)$ retornará *FALSE*; afinal, a soma não é uma operação precedente da multiplicação.

Perfeito. Agora já sabemos identificar as precedências, o que é muito importante para esse algoritmo. Nesse momento, faz-se necessário explicar que, para analisar uma expressão, precisamos percorrê-la caractere por caractere e, ao fazer isso, usar os caracteres encontrados como fonte de dados para um vetor de caracteres e uma pilha. É nesse vetor de caracteres que vamos encontrar o final de nossa expressão em formato *posfixo*; já a pilha é

uma estrutura de dados auxiliar para que possamos organizar corretamente os operadores e também os parênteses (quando eles existirem nas expressões analisadas).

Nesse momento, usaremos uma expressão sem os parênteses, tal como a expressão inicialmente apresentada: $X + Y * Z$. A [Tabela 10.1](#) representa, passo a passo, como vamos manipular caractere por caractere e onde os colocaremos para encontrar no final a nossa expressão *posfixa*.

TABELA 10.1: Passo a passo para transformar uma expressão infixa em posfixa (neste caso, sem parênteses e com precedência de operador já resolvida)

	Símbolo	String	Pilha
1	X	X	
2	+	X	+
3	Y	X Y	+
4	*	X Y	+ *
5	Z	X Y Z	+ *
6		X Y Z *	+
7		X Y Z * +	

Observe que, ao ler um símbolo da expressão, caso ele seja um operando (linhas 1, 3 e 5 da [Tabela 10.1](#)), é imediatamente encaminhado para a *string*, que formará a expressão *posfixa* final (linha 7). Contudo, se for um operador, esse símbolo é colocado na pilha (linhas 2 e 4), ou seja, o operador é empilhado usando a função *push* de pilha. Ao final, o que está na pilha é desempilhado usando o *pop* e jogado na *string* de saída, finalizando a expressão posfixa.

Fácil né? Foi mesmo, mas vamos complicar um pouco. Nada de mais, mas vai requerer um pouco mais de atenção. Propomos que usemos a seguinte expressão: $X / (Y - Z)$. Qual será a expressão *posfixa* dessa infixa? Será que você conseguiria

rascunhar sozinho a tabela que vamos apresentar a seguir? Acredito que sim, mas antes devemos lhe explicar uma nova regra, pois nessa expressão temos parênteses, e isso muda algumas coisas.

Assim que você encontrar um “abre parêntese”, basta empilhá-lo como se fosse um operando; porém, ao encontrar um “fecha parêntese” terá que desempilhar até achar um “abre parêntese” (isto é, desempilhar o “abre parêntese” que está empilhado).

Agora tente fazer essa tabela.

Apresentamos a [Tabela 10.2](#) como uma sugestão de solução para a conversão de infixo para *posfixo* da expressão $X / (Y - Z)$.

Só para terminar essa breve introdução sobre a utilização das pilhas em análise de expressões, por que não atribuímos valores para os operandos? Por exemplo, $X = 10$, $Y = 5$ e $Z = 3$. A expressão $X Y Z - /$ será percorrida caractere por caractere, da esquerda para a direita, até encontrar um operador; quando isso ocorre, são pegos os dois operandos imediatamente antes desse operador, e é feita a operação matemática envolvida. O primeiro operando é o sinal de subtração – estamos falando de $X Y Z - /$, nesse caso, $5 - 3$, resultando no valor 2, e a expressão é

TABELA 10.2: Conversão de infixo para posfixo da expressão $X / (Y - Z)$

	Símbolo	String	Pilha
1	X	X	
2	/	X	/
3	(	X	/ (
4	Y	X Y	/ (
5	-	X Y	/ (-
6	Z	X Y Z	/ (-
7	)	X Y Z -	/
8		X Y Z - /	

Observe que o “(” e “-” foram desempilhados. O operador “-” foi para a String, o parêntese foi apenas consumido.

atualizada: X^2 / , agora é só realizar essa operação, ou seja, $10 / 2$, portanto, o resultado dessa expressão com esses valores hipotéticos seria igual a 5. Pois bem, atribua valores a X Y e Z e brinque um pouco com essa expressão.



Vamos programar

Vimos vários conceitos que permeiam a pilha como estrutura de dados. O que você acha de olharmos esses mesmos conceitos, só que aplicados a outras linguagens bastante importantes? Divirta-se com os códigos em Java e Python.

Java

Código 10.1

```
private static void push(Stack st, int info)
{
    try
    {
        st.push(info);
    }
    catch (Exception e)
    {
        System.out.println("Pilha cheia, impossivel INSERIR elemento! Erro: "+e.toString() );
    }
}
```

Código 10.2

```
private static int pop (Stack st)
{
    try
    {
        return (int) st.pop();
    }
    catch (Exception e)
    {
        System.out.println("Pilha vazia, impossivel RETIRAR elemento! Erro: "+e.toString() );
    }
    return -1;
}
```

Código 10.3

```
private static void imprimir(Stack st)
{
    System.out.println("Stack = "+st);
}
```

Código 10.4

```

public static void main(String[] args)
{
    Stack stack = new Stack();

    //sequência de inserções de valores na pilha
    push(stack, 10);
    imprimir(stack);
    push(stack, 20);
    imprimir(stack);
    push(stack, 50);
    imprimir(stack);
    push(stack, 30);
    imprimir(stack);

    //sequência de retiradas dos valores da pilha
    pop(stack);
    imprimir(stack);
    pop(stack);
    imprimir(stack);
    pop(stack);
    imprimir(stack);
    pop(stack);
    imprimir(stack);
    pop(stack);
    imprimir(stack);
}

```

Python

Código 10.1

```

def push(self, info):
    if self.topo < TAM:
        print "Inserir : %d" % (info)
        self.informacao[self.topo] = info # a variavel "topo" aponta para a posicao onde a nova informacao deve ser
                                           # inserida, portanto, devemos fazer a atribuicao.
        self.topo+=1 #atualiza a variavel que controla o "topo" para que se possa receber a proxima informacao
    else:
        print "Pilha cheia"

```

Código 10.2

```

def pop(self):
    if self.topo > 0: #Verifica se existe alguma informacao a ser retirada na pilha
        self.topo-=1 #atualiza o topo para a posicao onde temos a ultima informacao inserida
        valor = self.informacao[self.topo] #obtem a informacao
        self.informacao[self.topo]=-1 #atualiza a posicao com a informacao que nao representa nada
        print "Retirar : %d" % (valor)
        return valor #retorna o valor para quem chamou a funcao pop
    else:
        print "Pilha vazia"
        return -1

```

Código 10.3

```
def imprimir(self):
    print "Pilha"
    #repeticao entre as posicoes que contem informacoes para que sejam impressas na tela
    i = 0
    while i < self.topo:
        print "%d" % (self.informacao[i])
        i+=1
```

Código 10.4

```
#DEFINICAO DA ESTRUTURA
class Pilha:
    def __init__(self):
        self.informacao = [-1]*TAM
        self.topo = 0
        .
        .
        .
    #funções: push, pop e imprimir aparecem aqui
    .
    .
    .
    pilha = Pilha()
    //sequência de inserções de valores na pilha
    push();
    imprimir();
    push(20);
    imprimir();
    push(50);
    imprimir();
    push(30);
    imprimir();

    //sequência de retiradas dos valores da pilha
    pop();
    imprimir();
    pop();
    imprimir();
    pop();
    imprimir();
    pop();
    imprimir();
    pop();
    imprimir();
```



Para fixar

1. Faça um teste de mesa nas funções *push* e *pop*, levando em consideração a sequência abaixo. Ao final, escreva o que fica na fila:
 - a. entrada do número 2;
 - b. entrada do número 8;

- c. entrada do número 6;
 - d. retirar um número;
 - e. entrada do número 1;
 - f. retirar um número.
2. Construa um programa que auxilie na criação dos adesivos que são colocados em carros especiais, tal como ambulâncias e carros de bombeiros, para serem olhados pelo retrovisor; por exemplo: AMBULANCIA $\longleftrightarrow$ AICNALUBMA
 3. Utilizando-se de pilhas, faça um procedimento para saber se existe o mesmo número de entradas dos caracteres *c* e *d*.



Para saber mais

As pilhas são primordiais em vários mecanismos computacionais. Um desses usos foi tema de nossa discussão nos parágrafos anteriores, isto é, sua utilização em compiladores para que possamos interpretar expressões matemáticas e fazer nossos programas funcionarem.

As pilhas também são amplamente utilizadas e importantes no *sistema operacional* de nossas máquinas. Com certeza, usamos e criamos *softwares*, que são modulados para que o sistema operacional possa controlar os módulos que estão chamando outros módulos. Ele simplesmente empilha essas funções, procedimentos ou métodos (dependendo do paradigma de programação utilizado pelo programador). Lembrar como as pilhas funcionam torna mais fácil entender por que o sistema operacional se utiliza delas para se organizar nesse momento: ao empilhar a chamada feita, quando ela termina, basta desempilhá-la e passar a usar o próximo topo.

Caso queira aprofundar o estudo das diversas formas de usar as pilhas para manipular expressões matemáticas, leia o Capítulo 2 do livro *Estrutura de dados usando C* (Tenenbaum, 2004).



Navegar é preciso

No link http://www.frontiernet.net/~prof_tcarr/StackMachine/Stack.html existe um simulador que mostra graficamente o que acontece com as funções *push* e *pop*. Embora nele existam outras funcionalidades, é muito interessante ver o funcionamento dessas duas funções.

Exercícios

1. Transforme as expressões a seguir em forma posfixa.
 - a. $((X + Y) * Z) + w$
 - b. $(X + (Y - Z)) * W$
 - c. $X * (Y + Z - W)$
 - d. $(X * Y) / (Z / W)$
 - e. $(X * Y) * (Z * W)$
2. Transforme as expressões a seguir em forma infixa.
 - a. $X Y -$
 - b. $X Y Z + -$
 - c. $X Y + Z *$
 - d. $X Y - Z + G H I - + +$
3. Implemente a função chamada de *prec*d, introduzida e explicada neste capítulo.

Glossário

Operadores: símbolos matemáticos que representam soma, subtração, multiplicação e outros.

Operandos: caracteres que simbolizam as variáveis que podem assumir valores.

PUSH: palavra usada pelos programadores para indicar que se deseja fazer uma inserção na pilha.

POP: outra palavra muito usada pelos programadores para indicar que se deseja retirar uma informação da pilha.

Referência bibliográfica

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.



QUE VEM DEPOIS

Das estruturas de dados apresentadas até o momento, chegou a vez de aprender a usar uma nova forma de pensar e armazenar informação. Nos próximos capítulos deste livro serão apresentadas as Árvores (Capítulo 11), Árvores n-árias (Capítulo 12), Árvores Balanceadas (Capítulo 13), além dos grafos (Capítulo 14). Bons estudos!

ÁRVORES

“ Com organização e tempo, acha-se o segredo de fazer tudo e bem feito. ”

PITÁGORAS

Quando aumenta o volume de itens a serem manipulados, é fundamental que eles sejam organizados de forma que sua manipulação seja eficiente.

OBJETIVOS DO CAPÍTULO

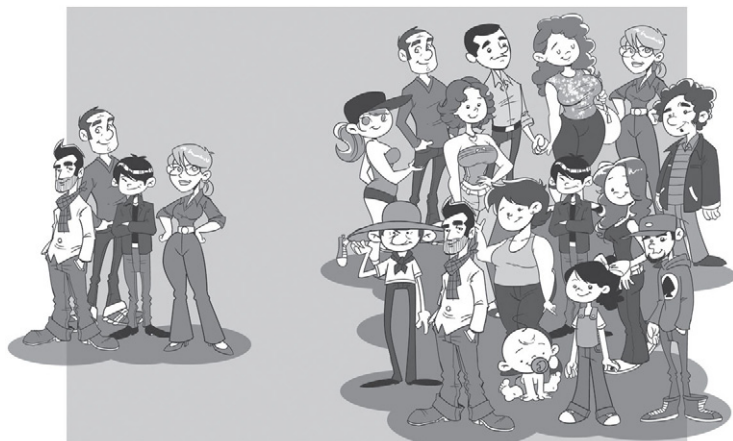
Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o motivo da existência da estrutura de dados árvore;
- conhecer os conceitos, terminologias utilizadas e a estruturação de dados na forma de árvore;
- criar e manipular dados na forma de árvores binárias.



Para começar

Sua tarefa: localizar o José da Silva e Souza nas imagens apresentadas.



Observando a primeira imagem, percebemos que não seria difícil localizar determinada pessoa dentre as ali apresentadas. Essa tarefa seria realizada com facilidade e em curto período de tempo. Por outro lado, o mesmo não pode ser dito em relação à segunda imagem. Localizar uma pessoa no meio da multidão não é tarefa fácil e nem mesmo poderia ser atendida num curto período de tempo.

Precisaríamos, de alguma forma, estabelecer uma organização na multidão para que a tarefa proposta fosse viável num tempo aceitável. Sabemos que uma das formas de organização mais antigas é a hierárquica. Na organização hierárquica da multidão poderíamos estabelecer critérios que permitissem agrupar as pessoas de acordo com determinadas características. Isso reduziria sensivelmente o espaço de busca, uma vez que não seria necessária a busca nos agrupamentos cujos integrantes não tivessem as características da pessoa procurada.

Outro exemplo seria localizar uma fotografia digital tirada numa viagem de férias feita há muitos anos e armazenada como arquivo num computador de uso pessoal, ou mesmo num repositório virtual. Imagine se todos os arquivos que você armazenou não estivessem organizados em pastas — seria quase impossível localizar a fotografia. É por esse motivo que os sistemas de arquivos são organizados de forma hierárquica.

Foi nesse contexto que surgiu a estrutura de dados denominada árvore. Nela, as informações são hierarquizadas segundo critérios que se mostram adequados a cada situação. Com a organização em árvores, é possível reduzir

de forma sensível o espaço de busca, tornando mais fáceis tarefas como as citadas.

Atenção



Árvores são estruturas em que os dados são dispostos de forma hierárquica. Nelas, os dados são armazenados em nós. Existe um nó principal, conhecido como *raiz*, a partir do qual surgem as ramificações, conhecidas como *subárvores*.

Para compreender melhor esse novo conceito, procure criar uma estruturação hierárquica na forma de árvore de diretórios (pastas) para suas fotografias digitais, de forma que seja fácil localizar determinada foto.

Dica



Para realizar essa tarefa, imagine como você procuraria uma fotografia. A partir disso, qual(is) critério(s) você consideraria para agrupar suas fotos em álbuns, pensando na facilidade futura para encontrá-la?

Conseguiu? Como ficou?

É provável que você tenha tido diferentes ideias, como, por exemplo, organizar as fotografias pelo ano e, dentro de cada ano, pelo mês, ou por tipo de relação social (família, amigos, colegas de trabalho, ou ainda por tipo de evento: férias, trabalho, lazer no cotidiano, festas etc.). Você pode ter misturado diferentes critérios, mas tudo bem. O mais importante é haver na hierarquia estabelecida uma forma de estruturação que facilite sua vida no futuro. Agora você já sabe o porquê da existência da estrutura de dados conhecida como *árvore*!!!

Neste capítulo, num primeiro momento, conheceremos o conceito, a terminologia e as formas de representação de árvores como estruturas de dados. Depois, estudaremos as árvores binárias, apresentando suas operações básicas. Vamos lá!



Conhecendo a teoria para programar

Como já descrito, a estrutura de dados *árvore* é uma forma de organização hierárquica em que os dados são armazenados em elementos conhecidos como *nós*. Portanto, um conjunto finito vazio (ou não) de nós compõe uma *árvore*.

Nessa estrutura existe um elemento principal, que é o *nó raiz*. Pode haver outros nós associados a ele, os quais seriam seus filhos ou descendentes. Os *nós filhos* formam subárvores do *nó pai* ou antecessor. Essas subárvores apresentam a mesma estrutura de árvore definida, ou seja, os nós filhos do nó raiz poderão ter seus próprios descendentes, e assim por diante.

Para ilustrar esse conceito, a [Figura 11.1](#) mostra a representação gráfica de uma árvore contendo nomes do nosso país, de alguns de seus estados e algumas cidades. Nessa figura temos como nó raiz aquele que contém o nome *Brasil*. Como nós filhos temos os *estados*, que, por sua vez, têm como filhos os nós contendo os *municípios*.

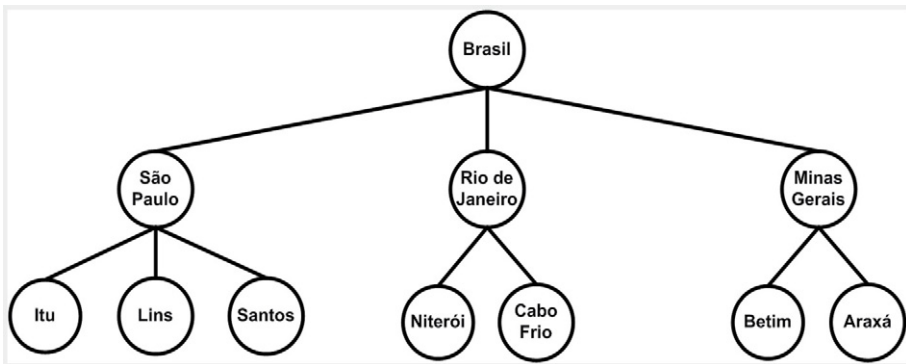


FIGURA 11.1: Árvore contendo país, estados e municípios.

Definição

Uma árvore A pode ser definida como:

1. estrutura vazia, $A = \{\}$;
2. conjunto finito e não vazio de nós, no qual existe um nó raiz R e nós que fazem parte de subárvores de A , ou seja, $A = \{R, A_1, A_2, A_3, \dots, A_n\}$.

Para ilustrar essa definição, vamos considerar a árvore A ([Figura 11.2](#)), em que temos a raiz B_0 e as subárvores A_1 , A_2 , A_3 e A_4 :

Conceitos

- **Nó raiz:** principal elemento da árvore, não apresenta ancestrais, e todos os nós da árvore são seus descendentes (diretos ou indiretos);
- **Nó pai:** elemento que apresenta descendentes na árvore, podendo ter um ou mais filhos;
- **Nó filho:** elemento que descende de algum outro nó da árvore, tendo apenas um nó pai;
- **Nó folha:** nó que não apresenta descendentes;

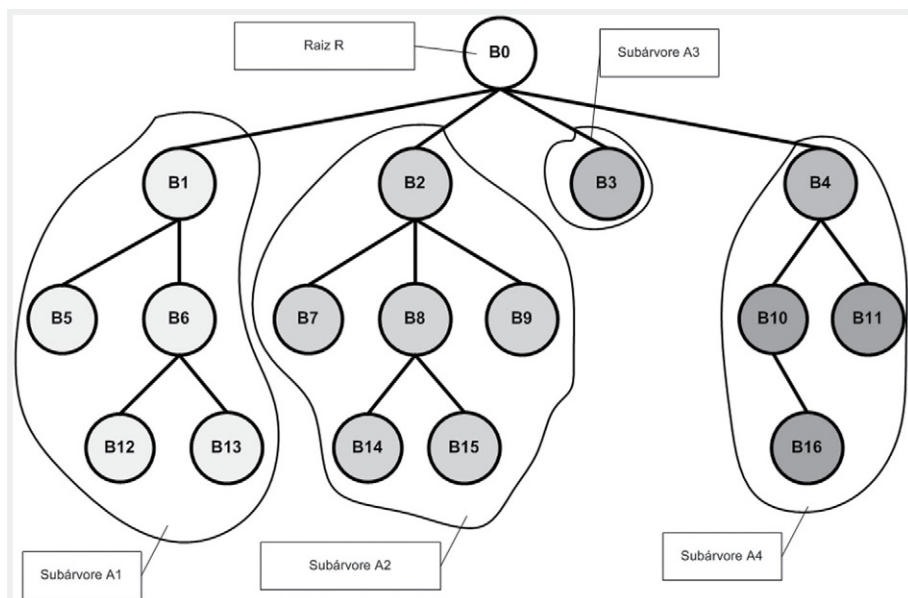


FIGURA 11.2: Árvore A.

- **Nível de um nó:** o nível do nó raiz é 0 ($N = 0$), e o nível dos nós restantes é igual ao nível do seu nó pai acrescido de 1. Para exemplificar, na [Figura 11.3](#), o nível de B0 é 0, B2 é 1, B5 é 2, e B16 é 3;

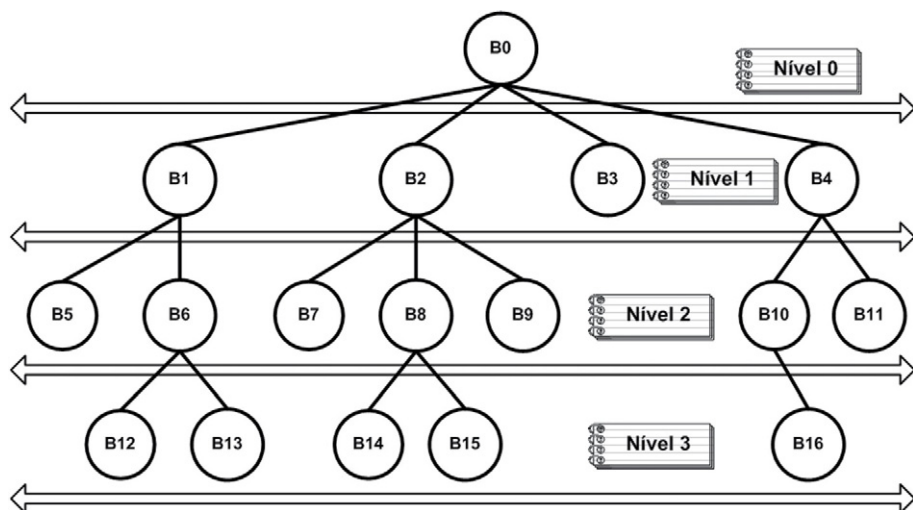


FIGURA 11.3: Nível de um nó da árvore.

- **Grau de um nó:** o grau de determinado nó é dado pelo número de seus filhos. Para exemplificar, na [Figura 11.4](#), o grau do nó B0 é 4, e o do nó B6 é 2;

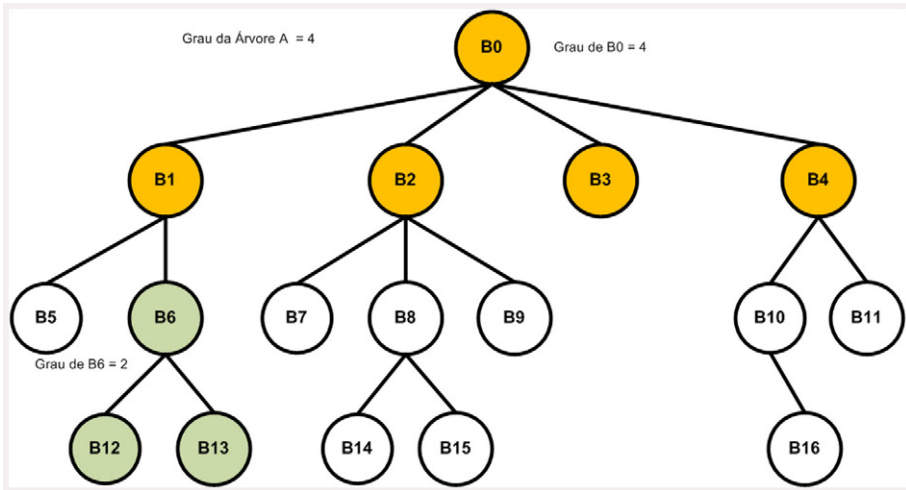


FIGURA 11.4: Grau de um nó e grau da árvore.

- **Grau da árvore:** uma árvore A tem grau igual ao grau máximo verificado para seus nós, ou seja, é igual ao do nó que apresenta mais filhos. Na árvore da [Figura 11.4](#) temos o grau da árvore A igual a 4;
- **Altura de um nó:** a altura de um nó B_i é igual à maior distância entre B_i e um nó folha que seja seu descendente. Para exemplificar, na [Figura 11.5](#) a altura de B_0 é 3, a de B_5 é 0, e a de B_{10} é 1;

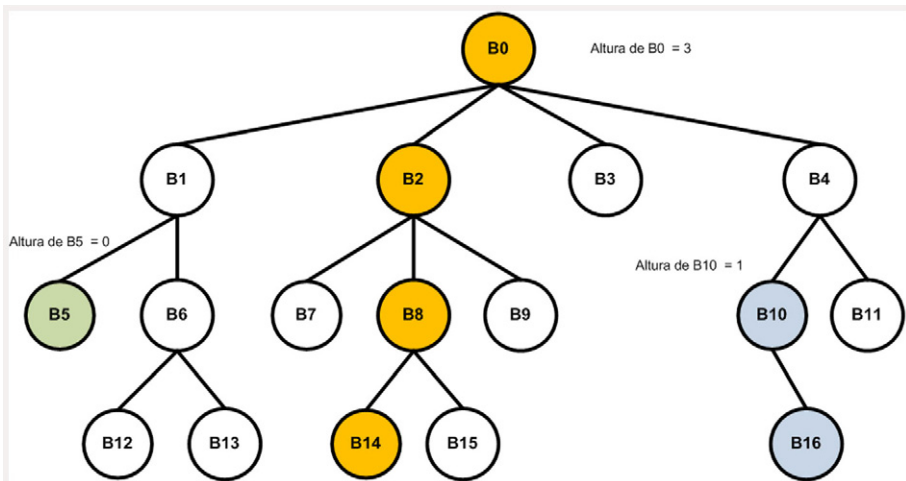


FIGURA 11.5: Altura de um nó.

- **Altura de uma árvore:** a altura de uma árvore corresponde à altura do nó raiz. Na árvore da [Figura 11.4](#) temos a altura da árvore A igual a 3.

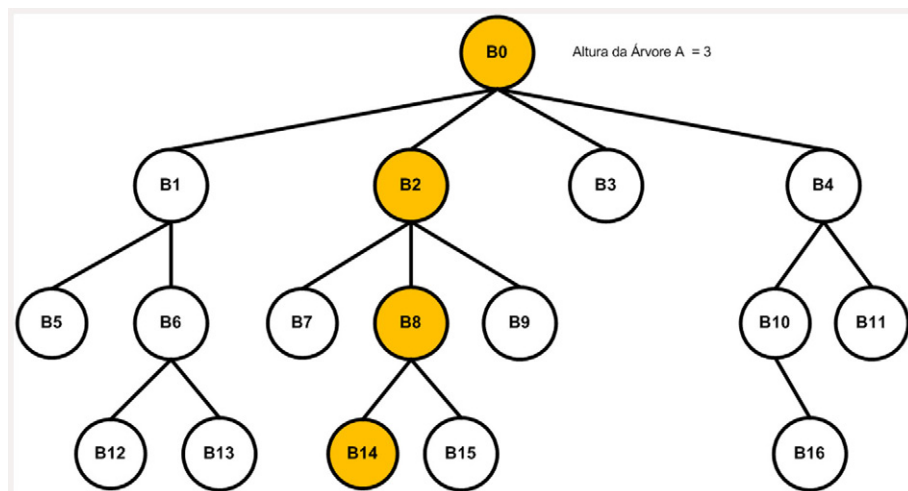


FIGURA 11.6: Altura da árvore.

Formas de representação

A forma gráfica de representação para a estrutura de dados árvore verificada na [Figura 11.2](#) é a mais utilizada e difundida. Porém, existem outras formas de representação que merecem destaque, como as mencionadas a seguir.

Diagrama de Venn (conjuntos aninhados)

O Diagrama de Venn foi criado em 1880, pelo lógico matemático inglês John Venn, para a representação de relações entre conjuntos. Consiste basicamente em criar círculos contendo todos os elementos do conjunto.

Ao considerarmos a estrutura de dados árvore um conjunto contendo o nó raiz e as subárvores formadas por seus descendentes, é fácil imaginar como seria a representação da árvore *A* ([Figura 11.2](#)) pelo Diagrama de Venn. Dentro do círculo que representaria a árvore *A*, teríamos o nó raiz *B0* e círculos representando as subárvores formadas por seus nós filhos. Dentro dos círculos de cada subárvore ficam o nó raiz dessas subárvores e novos círculos que representam as subárvores compostas pelos filhos da raiz da subárvore, e assim por diante, até chegarmos aos nós folhas. Na [Figura 11.7](#) temos a representação da árvore *A* por um Diagrama de Venn.

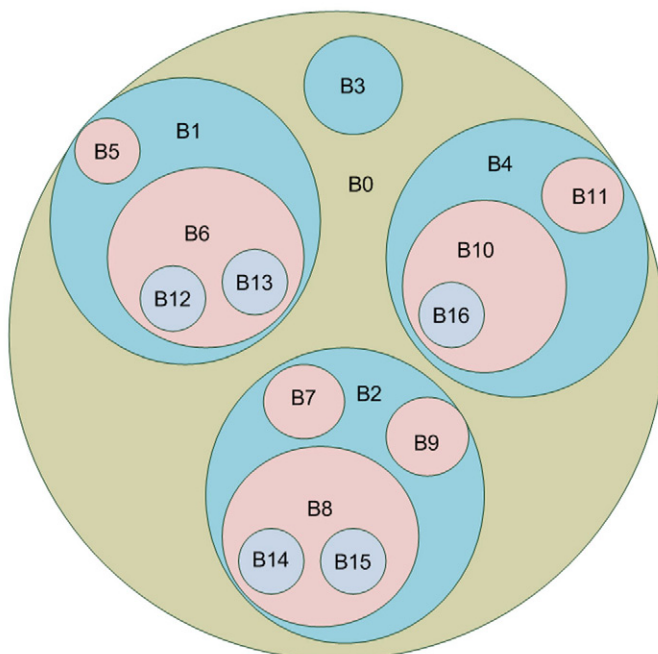


FIGURA 11.7: Árvore A representada por um Diagrama de Venn.

Parênteses aninhados

Essa forma de representação assemelha-se à observada no Diagrama de Venn. A diferença é que, nesse caso, utilizam-se parênteses em substituição aos círculos na delimitação do escopo dos conjuntos. Apesar de oferecer uma visualização inferior das relações entre os conjuntos, talvez uma vantagem dessa forma de representação esteja na possibilidade de representar a estrutura de dados árvore em uma única linha. A árvore A, representada por parênteses aninhados, ficaria:

```
(B0(B1(B5)(B6((B12)(B13)))(B2(B7)(B8(B14)(B15)))(B3)(B4(B10(B16))(B11))))
```

Atenção:

Na literatura, encontramos também os conceitos de *ordem de um nó* e *ordem de uma árvore*. O primeiro é usado por outros autores para representar o número (grau) mínimo de filhos de um nó, e alguns autores utilizam *ordem* como número máximo de filhos. Cuidado! O mesmo vale para *ordem de uma árvore*, que pode ser igual à *ordem do nó* com número mínimo ou máximo de filhos, dependendo do autor.



Árvores binárias

Numa estrutura de dados árvore, quando restringimos a dois o número máximo de filhos para um nó, temos uma *árvore binária*. Uma árvore binária

é caracterizada como um conjunto finito vazio (ou não) de nós. Esses nós são apresentados na forma de um nó raiz e seus descendentes, organizados em uma subárvore esquerda e uma subárvore direita.

Para melhor compreensão, observe a árvore binária da [Figura 11.8](#).

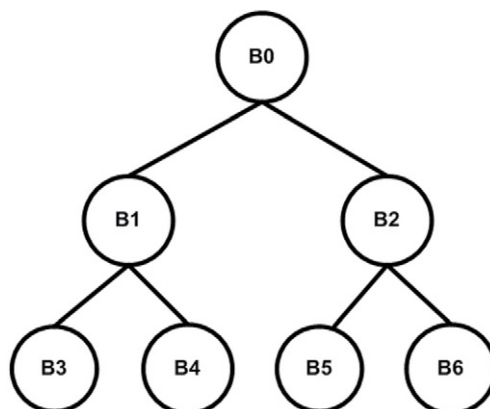


FIGURA 11.8: Árvore binária.

Nessa árvore temos B0 como nó raiz, B1 e B2 como nós filhos de B0, e assim por diante. Verificamos, também, que cada nó filho é um nó raiz da subárvore gerada a partir dele.

As árvores binárias herdam todos os conceitos existentes para a estrutura de dados árvore já vista, como: tipos de nó, nível de um nó ou árvore, grau de um nó ou árvore, e altura de um nó ou árvore. Da mesma forma, são aplicáveis as formas de representação estudadas.

Ao considerarmos que em determinado nível N de uma árvore binária temos K nós, teremos no máximo $2 \cdot K$ nós no nível imediatamente subsequente ($N + 1$). Dessa forma, como temos apenas 1 nó no nível 0, teremos no máximo 2 nós no nível 1, 4 no nível 2, e assim por diante.



Conceito

O número máximo de nós de uma árvore binária no nível N será 2^N . Por indução, é possível concluir que o número máximo de nós da árvore de nível T será $2^{T+1} - 1$.

Classificação para árvores binárias

Dependendo da distribuição dos nós em uma árvore binária, teremos as seguintes classificações: *árvore estritamente binária*, *árvore binária completa* e *árvore binária quase completa*.

Árvore estritamente binária

Nas situações em que, para todos os nós que não sejam nós folhas, não existem subárvores vazias, a árvore será denominada *árvore estritamente binária*. Um exemplo pode ser visto na [Figura 11.9](#).

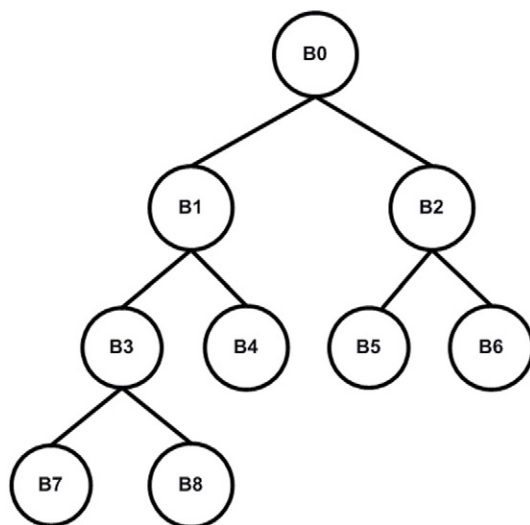


FIGURA 11.9: Árvore estritamente binária.

Árvore binária completa

Quando observamos a árvore da [Figura 11.9](#), percebemos que nem todos os nós folhas apresentam o mesmo nível. De outro lado, é possível que tenhamos uma árvore estritamente binária em que todos os nós folhas estejam no mesmo nível. Nessa situação temos uma árvore binária completa; veja a [Figura 11.10](#).

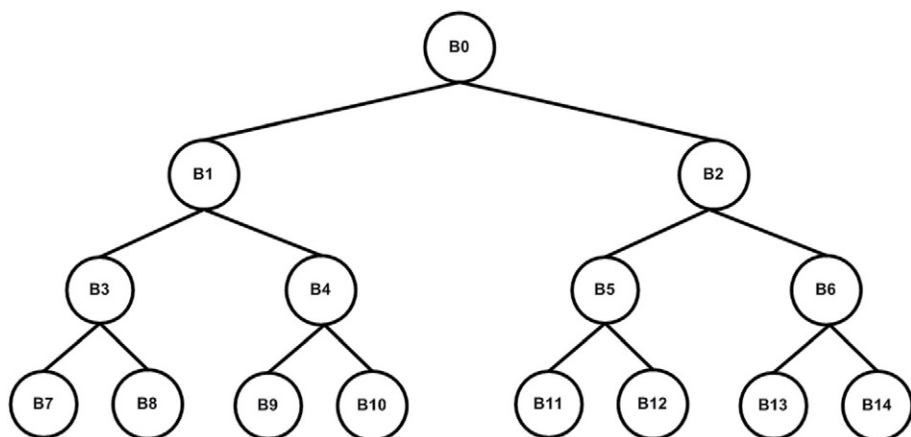


FIGURA 11.10: Árvore binária completa.

Árvore binária quase completa

Uma árvore binária que atender às condições a seguir será considerada árvore binária quase completa:

- todos os nós folhas estão no nível N ou $N-1$;
- para todo nó B_n que possuir um descendente direito no nível N (nível máximo da árvore), todo descendente esquerdo de B_n deverá ser nó folha no nível N .

Para compreender melhor, observe a [Figura 11.11](#). Na árvore (a) temos a caracterização de uma árvore binária quase completa; já na árvore (b) isso não ocorre, ou seja, ela não é uma árvore binária quase completa, por não atender à condição 2.

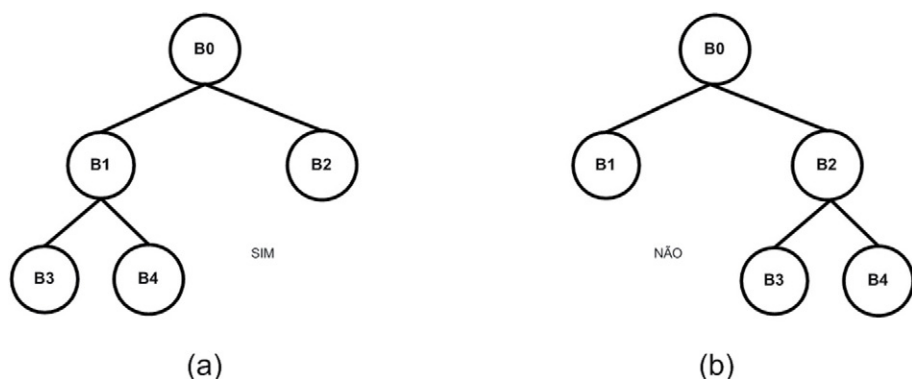


FIGURA 11.11: Caracterização de uma árvore binária quase completa.

Implementação de uma árvore binária

Para implementar uma árvore binária, existe a possibilidade de utilizar *alocação estática* ou *alocação dinâmica de memória*.

Implementação com alocação estática

O uso de alocação estática para a representação de uma árvore binária ocorre por meio da distribuição dos nós da árvore ao longo de um vetor (*array*). Essa distribuição ocorre da seguinte forma:

- o nó raiz fica na posição inicial do vetor (aqui considerada 0);
- para cada nó em determinada posição i do vetor, seu filho esquerdo ficará na posição $2*i + 1$, e seu filho direito ficará na posição $2*i + 2$.

Para ilustrar, veja na [Figura 11.12](#) como ficaria a árvore da [Figura 11.10](#) alocada num vetor.

Nos exemplos (I), (II) e (III) podemos notar que os filhos de B0 (posição 0) estão respectivamente nas posições 1 ($= 2*0 + 1$) e 2 ($= 2*0 + 2$), da

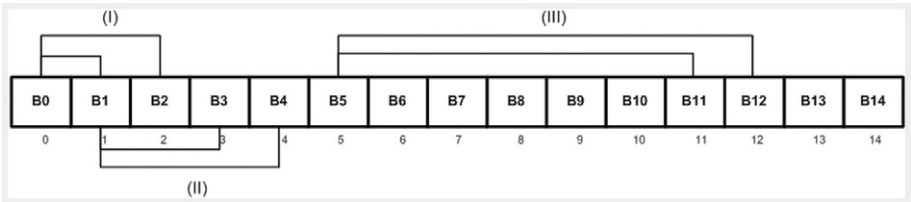


FIGURA 11.12: Árvore binária alocada num vetor.

mesma forma que os filhos de B1 estão nas posições 3 e 4 e os filhos de B5 nas posições 11 e 12.

Implementação com alocação dinâmica

O uso de alocação dinâmica para a representação de uma árvore binária ocorre através da definição de um registro contendo 3 campos básicos: um para armazenar o conteúdo do nó, uma para a ligação esquerda (nó filho à esquerda) e outro para a ligação direita (nó filho à direita), conforme a Figura 11.13.

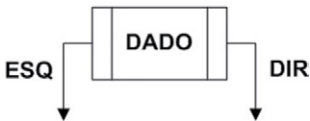


FIGURA 11.13: Representação de um nó alocado dinamicamente.

Para ilustrar, vejamos na Figura 11.14 como ficaria a árvore da Figura 11.10 alocada dinamicamente.

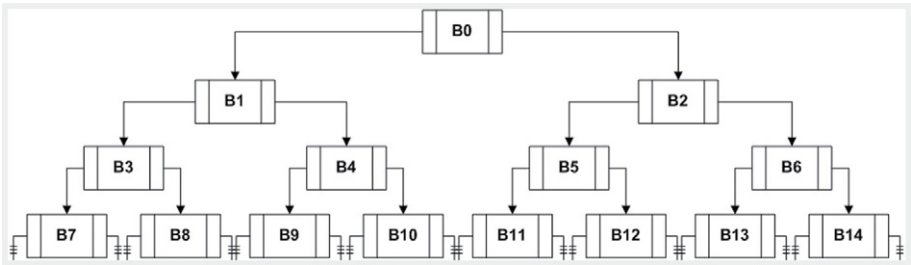


FIGURA 11.14: Representação de uma árvore binária alocada dinamicamente.

Dica

O uso de alocação estática tem como vantagem a simplicidade na manipulação de um vetor, mas, por outro lado, pode acarretar desperdício de espaço de armazenamento no caso de nós vazios. Já a alocação dinâmica se apresenta de forma oposta, com manipulação mais complexa da estrutura de armazenamento, mas eliminação do desperdício verificado na alocação estática.



Operações básicas numa árvore binária

A manipulação de uma árvore binária se dá através de diferentes operações possíveis. Neste livro, vamos focar as operações básicas de maior utilidade e difusão: *percurso* (também conhecido como *travessia* ou *varredura*), *inserção* e *remoção*.

Antes de começarmos a discutir e apresentar essas operações, precisamos apresentar formas de declarar e criar uma árvore binária, que, como descrito, pode ser feita de forma estática ou de forma dinâmica. A forma estática consiste em apenas declarar um vetor e manipulá-lo como descrito no item “Implementação com Alocação Estática”, porém, devido ao alto desperdício de memória provocado por essa forma, ela não é a mais utilizada. Resta-nos, portanto, fazer uso da forma dinâmica, que, deste ponto em diante, será a forma padrão adotada.

Declaração de uma árvore binária

Consiste na definição do nó por meio de um registro. Os campos que compõem esse registro são: *dado*, que armazena o(s) elemento(s) de dado(s) do nó; *esq*, que armazena a ligação com o nó filho à esquerda; e *dir*, que armazena a ligação com o nó filho à direita.

Código 11.1

```
typedef struct tipo_no no;
struct tipo_no
{
    tipo_dado dado;
    struct tipo_no *esq;
    struct tipo_no *dir;
};
```

Alguns autores utilizam também um campo adicional, denominado *pai*, que armazena a ligação com o nó pai. Esse campo pode ser descartado se a forma de percorrer a árvore for da raiz em direção às folhas. Por outro lado, se a forma adotada for das folhas para a raiz, os campos *esq* e *dir* é que serão descartados. Neste livro, adotaremos o percurso da raiz para as folhas, por isso a ausência do campo *pai* no registro que representa o nó.

Para facilitar a compreensão dos conceitos, nos exemplos sobre árvores vamos adotar que *tipo_dado* será um valor do tipo *inteiro*. Assim:

```
typedef int tipo_dado;
```

Percurso de árvore binária

Quando desejamos apresentar todos os elementos de uma árvore binária (independentemente de existir ou não alguma ordenação interna entre os elementos), é necessário estabelecermos um padrão para o percurso da árvore. No caso das árvores binárias, encontramos três formas básicas de percurso: *pré-ordem*, *em-ordem* e *pós-ordem*.

A distinção entre as formas de percurso se dá pela ordem em que os nós são visitados.

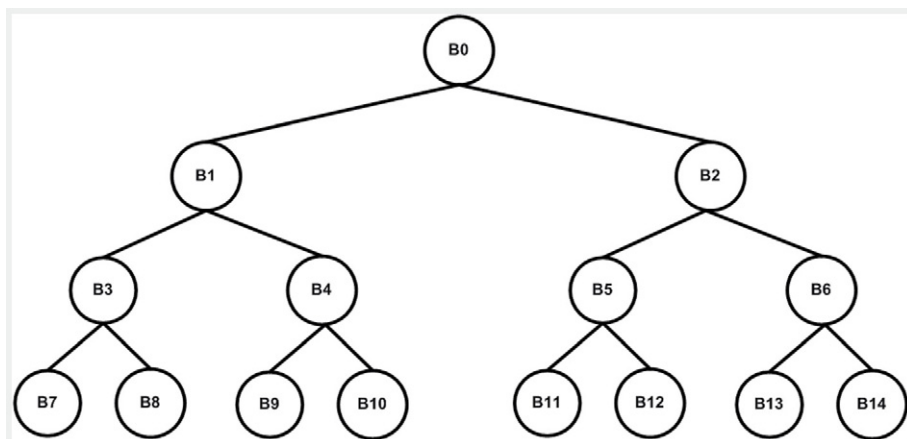
Percurso em pré-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita).

Para ilustrar, tomemos novamente o exemplo da árvore binária apresentada na [Figura 11.10](#) e vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pré-ordem.

PERCURSO PRÉ-ORDEM:



B0-B1-B3-B7-B8-B4-B9-B10-B2-B5-B11-B12-B6-B13-B14

A seguir, tem-se a implementação da função `pre_ordem`, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.2

```
void pre_ordem(no *Raiz)
{
    if (Raiz != NULL)
    {
        printf("%d ", Raiz->dado);
        pre_ordem(Raiz->esq);
        pre_ordem(Raiz->dir);
    }
}
```

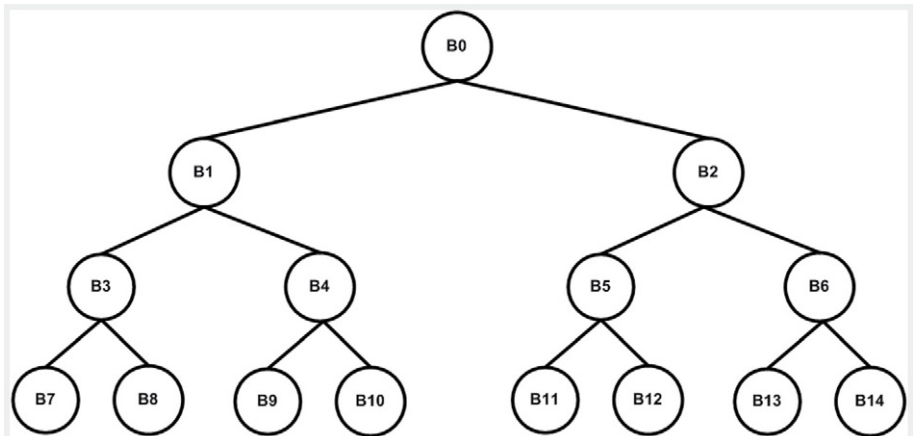
Percurso em em-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- apresenta o elemento do nó visitado;
- passa para o elemento do nó filho à direita (subárvore à direita).

Reproduzindo novamente a árvore da [Figura 11.10](#), vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso em-ordem.

PERCURSO EM-ORDEM:



B7-B3-B8-B1-B9-B4-B10-B0-B11-B5-B12-B2-B13-B6-B14

A seguir, tem-se a implementação da função `em_ordem`, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.3

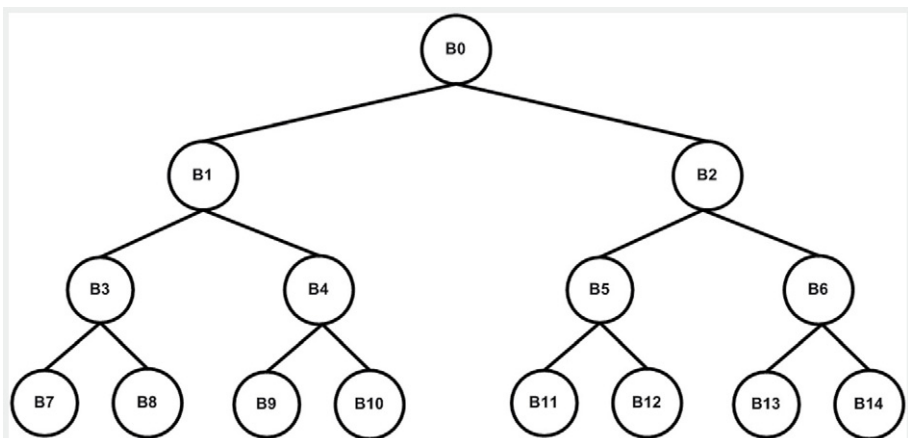
```
void em_ordem(no *Raiz)
{
    if (Raiz != NULL)
    {
        em_ordem(Raiz->esq);
        printf("%d ", Raiz->dado);
        em_ordem(Raiz->dir);
    }
}
```

Percurso em pós-ordem

Nessa forma de percurso, os nós de uma árvore binária são visitados de forma recursiva na seguinte ordem:

- passa para o elemento do nó filho à esquerda (subárvore à esquerda);
- passa para o elemento do nó filho à direita (subárvore à direita);
- apresenta o elemento do nó visitado.

Usando mais uma vez a reprodução da árvore da [Figura 11.10](#), vejamos como ficaria a ordem de apresentação dos elementos dessa árvore num percurso pós-ordem.

PERCURSO PÓS-ORDEM:

B7-B8-B3-B9-B10-B4-B1-B11-B12-B5-B13-B14-B6-B2-B0

A seguir, tem-se a implementação da função *pos_ordem*, que apresenta como saída os elementos da árvore binária, segundo os critérios já descritos.

Código 11.4

```
void pos_ordem(no *Raiz)
{
    if (Raiz != NULL)
    {
        pos_ordem(Raiz->esq);
        pos_ordem(Raiz->dir);
        printf("%d ", Raiz->dado);
    }
}
```

Inserção numa árvore binária

Quando abordamos a inserção numa árvore binária, é possível adotar diferentes formas de preenchimento. Um exemplo seria inserir novos elementos da esquerda para a direita, preenchendo a árvore por nível, ou seja, escolhendo sempre o primeiro nó livre (vazio) mais à esquerda no nível ainda não totalmente preenchido. A dificuldade, nessa forma de preenchimento, é que não existe qualquer organização dos dados (ordenação) que permita, posteriormente, uma busca mais eficiente por determinado elemento pertencente à árvore.

Assim, para se usar a árvore binária como estrutura que auxilie na busca (*árvore binária de busca*), adotou-se que, para cada elemento da árvore, existe uma *chave* associada ao mesmo, além dos *dados secundários* desse elemento. Essa *chave* será a responsável por promover a ordenação necessária entre os elementos da árvore.

No contexto de árvores binárias de busca, adotou-se também que os elementos com chaves *menores* que a contida num elemento que está armazenado no nó *N* da árvore serão inseridos como descendentes à *esquerda* de *N* na árvore. Consequentemente, os elementos com chaves *maiores* serão inseridos como descendentes à *direita* de *N*.

Atenção

Lembramos aqui que, quando manipulamos uma árvore binária como uma estrutura de dados que ofereça uma forma de organização ordenada dos dados, na verdade estamos nos referindo a uma árvore binária de busca, que, por motivo de simplificação, neste livro estaremos denominando apenas árvore binária.

Para facilitar a exemplificação dos conceitos, neste livro optamos por utilizar apenas o campo *chave* para caracterizar o elemento que será armazenado na árvore, desconsiderando a presença de dados secundários do elemento. Como consequência, temos que no campo *dado* (do registro que caracteriza um nó da árvore) vamos armazenar um valor que será também a própria chave de ordenação.

Consideremos o exemplo da [Figura 11.15](#), no qual desejamos inserir a chave (valor) 22 na árvore binária apresentada. Notemos que, na localização do ponto para inserção de 22, primeiro vemos que 22 é menor que 50 e, a seguir, que 22 é menor que 40 (subárvore à esquerda); por fim, vemos que 22 é maior que 15 (subárvore à direita). Como 15 não apresenta subárvore à direita, 22 é inserido como filho à direita de 15.

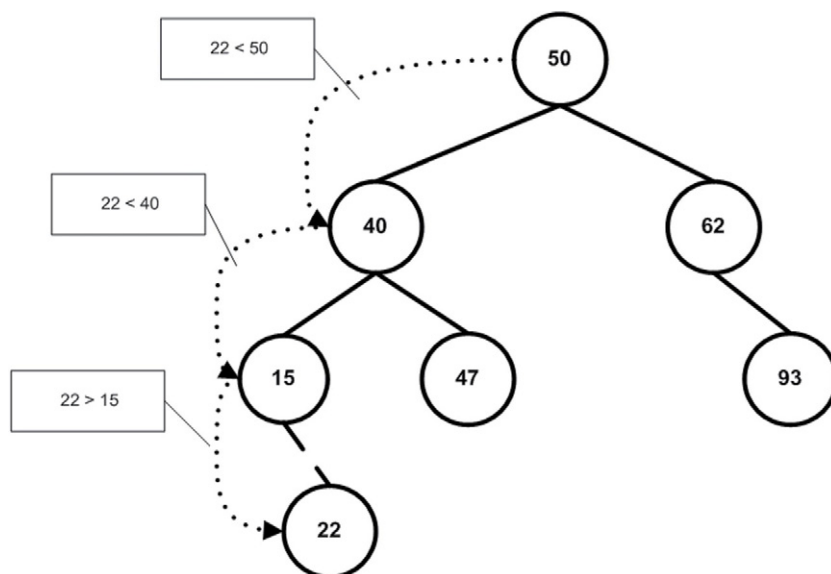


FIGURA 11.15: Inserção na árvore binária.

A seguir, tem-se a implementação da função *inserir_binaria*, que insere a chave *valor* na árvore binária.

Código 11.5

```
void inserir_binaria(no **Raiz, tipo_dado valor)
{
    no *E;

    if (*Raiz == NULL)                // Cria nó para inserção
    {
        E = (no *)malloc(sizeof(no)); // Aloca espaço na memória correspondente ao nó E
        E->dado = valor;               // insere o conteúdo (chave) do nó E
        E->esq = NULL;                // inicializa a subárvore esquerda com vazio
        E->dir = NULL;                // inicializa a subárvore direita com vazio
        *Raiz = E;
    }
    else if (valor < (*Raiz)->dado)
        inserir_binaria(&(*Raiz)->esq, valor);
    else if (valor > (*Raiz)->dado)
        inserir_binaria(&(*Raiz)->dir, valor);
    else
        printf("Elemento ja existente\n");
}
```

Ao observarmos a implementação da função *inserir_binaria*, podemos notar que se trata de um *procedimento recursivo*, em que a posição de inserção é procurada baseando-se na organização já descrita, e em que as chaves menores ficam sempre à esquerda e as maiores à direita de determinado nó da árvore (subárvore). Dessa forma, se a chave procurada for menor que a existente no nó investigado, é feita uma chamada recursiva de *inserir_binaria*, em que, como raiz, é passado o nó raiz da subárvore à esquerda. De forma análoga, se a chave for maior, a chamada ocorre para a subárvore à direita.

Remoção numa árvore binária

Para apresentar a operação de remoção numa árvore binária, adotaremos o mesmo padrão verificado na inserção, ou seja, nós filhos à esquerda contêm chaves menores do que a encontrada no nó pai, consequentemente, nós filhos à direita contêm chaves maiores que a existente no nó pai.

Diferentemente da inserção, a remoção traz dificuldades adicionais para sua operacionalização. Quando removemos um elemento, para que a organização prévia seja mantida, torna-se necessária a reorganização da árvore binária.

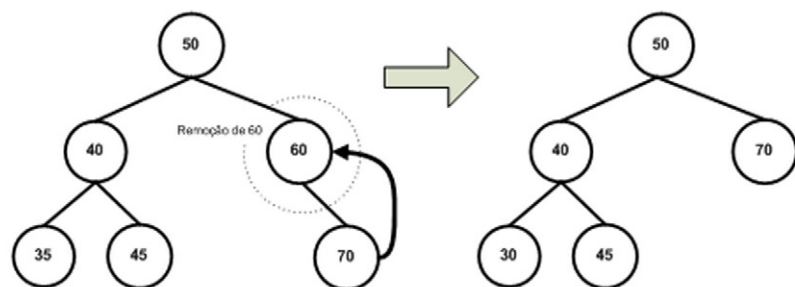
Dependendo da posição do elemento removido, diferentes ações podem ser necessárias. De forma geral, há três situações possíveis quanto ao nó em que se encontra o elemento a ser removido:

- **nó não apresenta subárvore à esquerda:** basta fazer com que o nó filho à direita passe a ser o nó pai;
- **nó não apresenta subárvore à direita:** basta fazer com que o nó filho à esquerda passe a ser o nó pai;
- **nó apresenta subárvores à esquerda e à direita:** deve ser deslocado, para a posição em que se encontra o nó Bi a ser removido, o nó com o elemento de maior chave da subárvore à esquerda de Bi.*

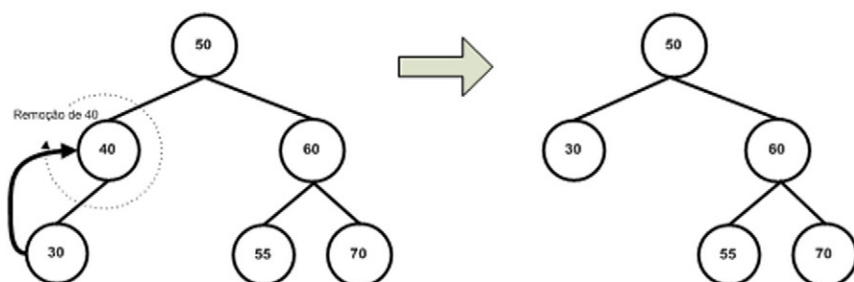
Para exemplificar, observemos os exemplos da [Figura 11.16](#). Na parte (a) temos a remoção da chave (valor) 60 correspondendo à situação (1), descrita anteriormente. Na parte (b), a remoção da chave 40 corresponde à situação (2) e, por fim, na parte (c), a remoção da chave 38 corresponde à situação (3).

A seguir são apresentadas as funções que permitem a remoção de um elemento da árvore binária. A função *remover_binaria* é a responsável por localizar e remover o elemento desejado. Porém, na situação descrita, quando o elemento a ser removido está num nó com subárvores à direita e à esquerda, torna-se necessário o uso da função *retornar_maior*, que tem como finalidade devolver o elemento de maior chave da subárvore esquerda, que ficará no lugar do elemento removido.

*Alguns autores adotam para a situação 3 o deslocamento, para a posição em que se encontra o nó Bi a ser removido, do nó com o elemento de menor chave da subárvore à direita de Bi.



(a)



(b)

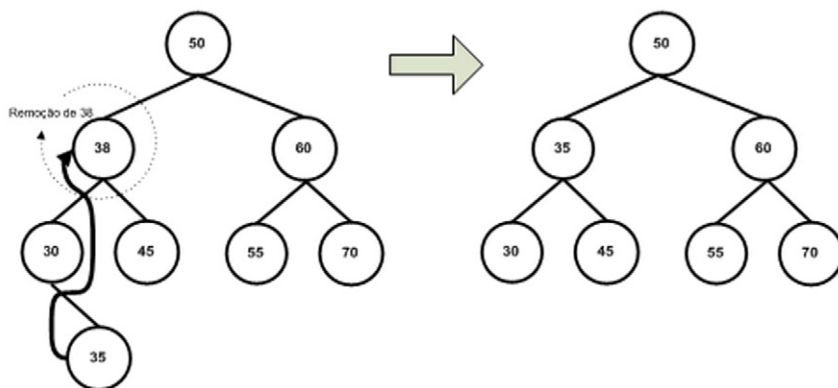


FIGURA 11.16: Remoções de elementos em árvores binárias.

Código 11.6

```

tipo_dado retornar_maior(no **X)
{
    tipo_dado valor;
    no *Aux;

    if ((*X)->dir != NULL)
        return(retornar_maior(&(*X)->dir)); // Continua procurando pelo maior elemento (chave) da subárvore
    else {
        Aux = *X;
        valor = (*X)->dado; // Retorna o maior elemento (chave) da subárvore analisada
        *X = (*X)->esq;      // Filho à esquerda passa a ser o pai, uma vez que o seu pai deixou de existir por ter sido
                             // deslocado para o lugar do nó que foi removido
        free(Aux);           // Libera o espaço alocado na memória para o elemento deslocado
        return(valor);
    }
}

void remover_binaria(no **Raiz, tipo_dado valor)
{
    no *Aux;
    if (*Raiz != NULL)
    {
        if ((*Raiz)->dado == valor)
        {
            Aux = *Raiz;
            if ((*Raiz)->esq == NULL) // Se a subárvore esquerda é vazia
            {
                *Raiz = (*Raiz)->dir; // Transforma o filho à direita em nó pai
                free(Aux);             // Libera o espaço alocado na memória para o elemento removido
            }
            else if ((*Raiz)->dir == NULL) // Se a subárvore direita é vazia
            {
                *Raiz = (*Raiz)->esq; // Transforma o filho à esquerda em nó pai
                free(Aux);             // Libera o espaço alocado na memória para o elemento removido
            }
            // Caso o nó a ser removido tenha subárvores à direita e esquerda, então coloca ali o maior elemento (chave) da subárvore à esquerda
            else (*Raiz)->dado = retornar_maior(&(*Raiz)->esq);
        }
        else if ((*Raiz)->dado < valor) // elemento (chave) procurado está na subárvore à direita
            remover_binaria(&(*Raiz)->dir, valor);
        else if ((*Raiz)->dado > valor) // elemento (chave) procurado está na subárvore à esquerda
            remover_binaria(&(*Raiz)->esq, valor);
    }
    else printf ("Elemento inexistente!\n"); // elemento (chave) procurado não faz parte da Árvore Binária
}

```

Eficiência na busca numa árvore binária

As operações básicas numa árvore binária (de busca) tem tempo proporcional à sua altura. Como a altura da árvore dependerá da quantidade N de chaves e de sua ordem de inserção na árvore, o tempo de resposta das operações básicas dependerá da quantidade e da distribuição das chaves pelas subárvores (subárvores com diferentes alturas). No pior caso em termos de altura da árvore binária, teríamos as inserções das chaves de forma que a altura da árvore fosse $N-1$, ou seja, um tempo de execução das operações básicas $O(N)$.*

Numa situação ideal de distribuição de N chaves pelas subárvores (árvore binária completa), porém, as operações básicas, no pior caso, seriam executadas num tempo $O(\log_2 N)$. Para aprofundamento sobre a eficiência de uma árvore binária, recomendamos a leitura do material mencionado na seção “Para Saber Mais” deste capítulo.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação Java e Python.

Java

Código 11.1

```
public class ArvoreBinaria
{
    private No raiz;
    private ArvoreBinaria arvoreEsquerda;
    private ArvoreBinaria arvoreDireita;

    public ArvoreBinaria() { }

    public ArvoreBinaria getArvoreDireita() {
        return arvoreDireita;
    }

    public void setArvoreDireita(ArvoreBinaria arvoreDireita) {
        this.arvoreDireita = arvoreDireita;
    }

    public ArvoreBinaria getArvoreEsquerda() {
        return arvoreEsquerda;
    }

    public void setArvoreEsquerda(ArvoreBinaria arvoreEsquerda) {
        this.arvoreEsquerda = arvoreEsquerda;
    }

    public No getRaiz() {
        return raiz;
    }

    public void setRaiz(No raiz) {
        this.raiz = raiz;
    }

    public class No {
        private Dado dado;

        public No(Dado dado) {
            this.dado = dado;
        }

        public Dado getDado() {
            return dado;
        }

        public void setDado(Dado dado) {
            this.dado = dado;
        }
    }

    public class Dado {
        private int valor;

        public Dado(int valor) {
            this.valor = valor;
        }

        public int getValor() {
            return valor;
        }

        public void setValor(int valor) {
            this.valor = valor;
        }
    }
}
```

*Em termos de eficiência em algoritmos, a notação $O(\dots)$ representa a ordem associada ao que se deseja mensurar. Assim, $O(N)$ representa uma ordem linear e $O(\log)$ uma ordem logarítmica.

Código 11.2

```
public void pre_ordem() {  
    if (this.raiz == null) {  
        return;  
    }  
  
    System.out.println("Valor: " + this.raiz.getDado().getValor());  
  
    if (this.arvoreEsquerda != null) {  
        this.arvoreEsquerda.pre_ordem();  
    }  
  
    if (this.arvoreDireita != null) {  
        this.arvoreDireita.pre_ordem();  
    }  
}
```

Código 11.3

```
public void em_ordem() {  
    if (this.raiz == null) {  
        return;  
    }  
  
    if (this.arvoreEsquerda != null) {  
        this.arvoreEsquerda.em_ordem();  
    }  
  
    System.out.println("Valor: " + this.raiz.getDado().getValor());  
  
    if (this.arvoreDireita != null) {  
        this.arvoreDireita.em_ordem();  
    }  
}
```

Código 11.4

```
public void pos_ordem() {
    if (this.raiz == null) {
        return;
    }

    if (this.arvoreEsquerda != null) {
        this.arvoreEsquerda.pos_ordem();
    }

    if (this.arvoreDireita != null) {
        this.arvoreDireita.pos_ordem();
    }

    System.out.println("Valor: " + this.raiz.getDado().getValor());
}
```

Código 11.5

```
public void inserir_binaria(int valor) {
    Dado dado = new Dado(valor);
    No no = new No(dado);
    inserir(no);
}

public void inserir(No no) {
    if (this.raiz == null) {
        this.raiz = no;
    } else {
        if (no.getDado().getValor() < this.raiz.getDado().getValor()) {
            if (this.arvoreEsquerda == null) {
                this.arvoreEsquerda = new ArvoreBinaria();
            }
            this.arvoreEsquerda.inserir(no);
        }
        else if (no.getDado().getValor() > this.raiz.getDado().getValor()) {
            if (this.arvoreDireita == null) {
                this.arvoreDireita = new ArvoreBinaria();
            }
            this.arvoreDireita.inserir(no);
        }
        else System.out.println("Elemento já existente");
    }
}
```


Código 11.6

```

public static ArvoreBinaria remover_binaria(ArvoreBinaria aux, int num) {
    ArvoreBinaria p, p2;
    if (aux.raiz.getDado().getValor() == num) {
        if (aux.arvoreEsquerda == aux.arvoreDireita) {
            return null;
        } else if (aux.arvoreEsquerda == null) {
            return aux.arvoreDireita;
        } else if (aux.arvoreDireita == null) {
            return aux.arvoreEsquerda;
        } else {
            p2 = aux.arvoreDireita;
            p = aux.arvoreDireita;
            while (p.arvoreEsquerda != null) {
                p = p.arvoreEsquerda;
            }
            p.arvoreEsquerda = aux.arvoreEsquerda;
            return p2;
        }
    } else if (aux.raiz.getDado().getValor() < num) {
        aux.arvoreDireita = remover_binaria(aux.arvoreDireita, num);
    } else {
        aux.arvoreEsquerda = remover_binaria(aux.arvoreEsquerda, num);
    }
    return aux;
}
}

```

Phyton

Código 11.1

```

class arvoreBinaria(object):
    def __init__(self, value=None, esq = None, dir = None):
        self.value = value
        self.esq = esq
        self.dir = dir

```

Código 11.2

```
def pre_ordem(self):
    if self.value <> None:
        print self.value
        if self.esq <>None:
            self.esq.pre_ordem()
        if self.dir <>None:
            self.dir.pre_ordem()
```

Código 11.3

```
def em_ordem(self):
    if self.value <> None:
        if self.esq <>None:
            self.esq.em_ordem()
        print self.value
        if self.dir <>None:
            self.dir.em_ordem()
```

Código 11.4

```
def pos_ordem(self):
    if self.value <> None:
        if self.esq <>None:
            self.esq.pos_ordem()
        if self.dir <>None:
            self.dir.pos_ordem()
        print self.value
```

Código 11.5

```
def inserir_binaria(self, item):
    if self.value == item:
        return
    else:
        if item < self.value:
            if self.esq != None:
                self.esq.inserir_binaria(item)
            else:
                self.esq = arvoreBinaria(item)
        else:
            if self.dir != None:
                self.dir.inserir_binaria(item)
            else:
                self.dir = arvoreBinaria(item)
```

Código 11.6

```
def remover_binaria(self, aux, num):
    p = arvoreBinaria()
    p2 = arvoreBinaria()
    if aux.value==num:
        if aux.esq==aux.dir:
            return None
        elif aux.esq==None:
            return aux.dir
        elif aux.dir==None:
            return aux.esq
        else:
            p2=aux.dir
            p=aux.dir
            while p.esq<>None:
                p=p.esq
    elif aux.value < num:
        aux.dir = remover_binaria(aux.dir,num)
    else:
        aux.esq = remover_binaria(aux.esq,num)
    return aux
```



Para fixar!

1. Considerando o que você estudou, pense numa situação do seu cotidiano em que a organização hierárquica em árvore possa ser usada. Procure estruturar os elementos que compõem a situação.
2. Usando as funções definidas neste capítulo, crie uma árvore binária contendo os valores (chaves) 50, 40, 15, 62, 93, 47, 35, 68, 10, 37 e 22. Apresente os percursos pré-ordem, em-ordem e pós-ordem para essa árvore. Tente remover o valor 100 (inexistente). Depois, remova os valores 93 e 40. Repita a apresentação dos percursos pré-ordem, em-ordem e pós-ordem para a nova árvore (após a remoção).
3. Procure na internet simuladores de árvores binárias e utilize-os com os conjuntos de dados que criou para o exercício 1. Depois, compare os resultados encontrados com os que você obteve usando as funções aqui apresentadas.



Para saber mais...

Dentro do propósito geral deste livro, que é apresentar de forma clara e numa linguagem acessível uma visão geral básica das principais estruturas de dados existentes, orientamos aqueles que desejam se aprofundar nos estudos sobre árvores binárias que leiam a Parte III (Capítulos 12 e 13) do livro *Algoritmos: teoria e prática* (Wirth, 1989).



Navegar é preciso

Existem diversas ferramentas disponíveis na internet que podem contribuir com o seu aprendizado sobre árvores binárias. Dentre elas citamos:

- ASTRAL (<http://www.ic.unicamp.br/~rezende/Astral.htm>);

Relaciona um conjunto de ferramentas executáveis gratuitas que permitem investigar, entre outras coisas, o funcionamento das árvores binárias.

- TBC_AED (http://algoritmo.dcc.ufba.br/~heitor/Projetos/TBC_AED_GRAFOS_WEB/TBC_AED_GRAFOS_WEB.html);

Oferece ferramentas WEB para o aprendizado de estrutura de dados, dentre as quais uma para árvore binária.

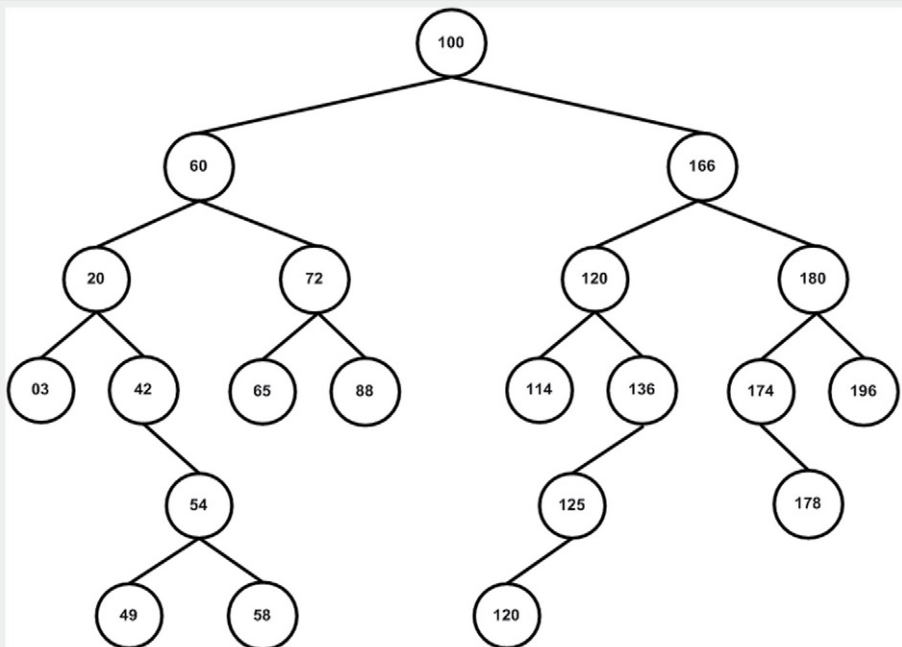
- SIMULED (<http://simuledufg.sourceforge.net/>);

Ferramenta com código em Java (GNU GPL) que permite a execução das operações mais comuns com estruturas de dados.

- Outras ferramentas em forma de Applets
[\(http://www.cs.umd.edu/~egolub/Java/BinaryTree.html\);](http://www.cs.umd.edu/~egolub/Java/BinaryTree.html)
[\(http://www.cosc.canterbury.ac.nz/mukundan/dsal/BSTNew.html\);](http://www.cosc.canterbury.ac.nz/mukundan/dsal/BSTNew.html)
[\(http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html\);](http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html)
[\(http://nova.umuc.edu/~jarc/idsv/lesson4.html\).](http://nova.umuc.edu/~jarc/idsv/lesson4.html)

Exercícios

1. Crie uma função de percurso em que os elementos da árvore da Figura 11.10 sejam apresentados por nível, ou seja, em que a ordem de apresentação seja B0-B1-B2-B3-B4-B5-B6-B7-B8-B9-B10-B11-B12-B13-B14.
2. Para a árvore binária a seguir, responda:



- a. A árvore foi construída corretamente?
- b. Que nós pertencem ao nível 3 da árvore?
- c. Qual é a altura do nó que contém o valor 120?
- d. Como ficaria a árvore com a inserção dos valores 59, 61 e 128?
- e. Como ficaria a árvore com a remoção dos valores 42, 166 e 174?

3. Crie uma função que devolva o nível de determinado nó que contenha o valor passado para a função.
4. Crie uma função que devolva a soma dos valores presentes nos *nós folhas* de uma árvore binária.

Conseguiu? Parabéns, você já domina os conceitos fundamentais da estrutura de dados árvore!!!

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Agora que você já conhece os conceitos fundamentais de árvores e, principalmente, de árvores binárias, vamos generalizar essas ideias considerando que cada nó poderia ter um número de filhos maior que os dois vistos nas árvores binárias. Será que teremos mudanças significativas? É o que você descobrirá no Capítulo 12!

Árvores N-árias

“Ciência é conhecimento organizado. Sabedoria é vida organizada.”

IMMANUEL KANT

A sabedoria em manter uma boa organização dos dados em uma estrutura de dados árvore, muitas vezes, leva à necessidade de novas escolhas. Ampliar o número de subárvores para um nó da árvore pode significar maior organização dos dados ali armazenados.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o motivo de ampliar o número de subárvores (nós filhos) de uma estrutura de dados árvore;
- criar e manipular dados na forma de árvores N-árias.



Para começar

Imagine que você tenha duas jabuticabeiras. Numa delas, cada galho, a partir da raiz, se divide em no máximo dois novos galhos; na outra, cada galho pode se dividir em até cinco novos galhos.

Considere ainda que os frutos nascem sempre ou na ponta dos galhos ou no ponto em que ele se divide em novos galhos. Na primeira jabuticabeira temos apenas um fruto na ponta de cada galho e um fruto nesse ponto de divisão dos novos galhos. Já na segunda temos quatro frutos, tanto na ponta de cada galho, como no ponto de divisão dos novos galhos.

A partir dessas considerações, qual seria o tamanho mínimo das duas jabuticabeiras para que cada uma delas comportasse 200 frutos?

No caso de você decidir colher os frutos, em qual delas haveria maior dificuldade?



Você deve ter concluído que, para comportar as 200 jabuticabas, a primeira jabuticabeira teria altura muito maior que a segunda, levando um tempo maior para que todos os frutos fossem alcançados e colhidos.

Considerando as restrições impostas, quanto maior o volume de frutos, maior deverá ser a altura das jabuticabeiras e maior a dificuldade na colheita, principalmente na primeira árvore. Na estrutura de dados árvore ocorre o mesmo.

Atenção



Conforme aumentamos o volume de dados a serem armazenados numa estrutura de árvore, maior é a dificuldade de manter a eficiência na recuperação, principalmente tendo um número pequeno de subárvores a partir de determinado nó da árvore.

Para compreender melhor esse novo conceito, procure repetir a tarefa proposta no início do Capítulo 11, em que lhe foi solicitado criar uma estruturação hierárquica na forma de árvore de diretórios (pastas) para suas fotografias digitais, de forma que fosse fácil localizar determinada fotografia. Porém, dessa vez, crie tal representação considerando a limitação de que, em cada pasta, somente é possível ter duas novas pastas dentro (como uma árvore binária).

Conseguiu? Procure comparar a estruturação que você construiu no Capítulo 11 (que provavelmente continha várias subpastas dentro de uma pasta) com a nova, relacionando possíveis vantagens e desvantagens de cada forma de estruturação.



Dica

Para estabelecer a comparação, considere a facilidade para manipular suas pastas e o tempo gasto para encontrar uma fotografia desejada em ambas as estruturações.

Depois disso, esperamos que você tenha entendido o motivo da existência das árvores em que não ficamos limitados a dois novos nós filhos (subárvores) a partir de cada nó, como ocorre com as árvores binárias. Essas novas formas de árvores recebem o nome de *árvores N-árias*.

Neste capítulo, num primeiro momento, conheceremos a conceituação de árvores *N*-árias. Depois, estudaremos suas operações básicas e, por fim, procuraremos caracterizar brevemente suas possíveis vantagens, apresentando também um exemplo de aplicação prática. Vamos lá!



Conhecendo a teoria para programar

As definições apresentadas no início do Capítulo 11 para a estrutura de dados árvore a caracterizam de forma geral, portanto, englobam as árvores *N*-árias. Assim, tanto a definição quanto os conceitos apresentados (Nó, nó

raiz, nó filho, nível de um nó ou árvore, grau de um nó ou árvore e altura de um nó ou árvore) são válidos para as árvores N -árias.

Entretanto, aqui nosso interesse recai sobre a árvore N -ária com organização interna que permita o estabelecimento de determinada ordem entre os valores atribuídos a seus elementos. Esse tipo de árvore é chamada *árvore N -ária de busca*.

Atenção

Assim como no Capítulo 11 tratamos as árvores binárias de busca apenas como árvores binárias, neste capítulo, quando fizermos menção a uma árvore N -ária, estaremos, na verdade, abordando uma árvore N -ária de busca.



Neste contexto, tal como nas árvores binárias (de busca), associa-se a cada elemento da árvore uma *chave*, através da qual se dá a ordenação desses elementos. Neste livro, optamos por utilizar apenas o campo *chave* para caracterizar o elemento que será armazenado na árvore, desconsiderando a presença de dados secundários do elemento. Assim, nos exemplos dados aqui, ao fazer uma referência à *chave*, referenciaremos o próprio valor do elemento. Para adicionar dados secundários ao elemento da árvore, bastaria criar no registro (que será caracterizado posteriormente neste capítulo) outros campos que possam armazená-los, sem qualquer outro impacto nos conceitos e ideias que serão discutidas.

Definição

Uma árvore N -ária (de busca) AN pode ser definida como:

- estrutura vazia, $AN = \{\}$;
- conjunto finito e não vazio de nós, em que existe um nó raiz R e nós que fazem parte de subárvores de AN , $AN = \{R, A1, A2, A3, \dots, An\}$, sendo que cada nó contém até $n-1$ elementos, sendo n o número máximo de subárvores a partir de um nó;
- os elementos dentro de um nó estão sempre ordenados por meio de de suas respectivas chaves;
- à esquerda de cada elemento Ei de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves menores que a chave de Ei ;
- à direita de cada elemento Ei de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves maiores que a chave de Ei .

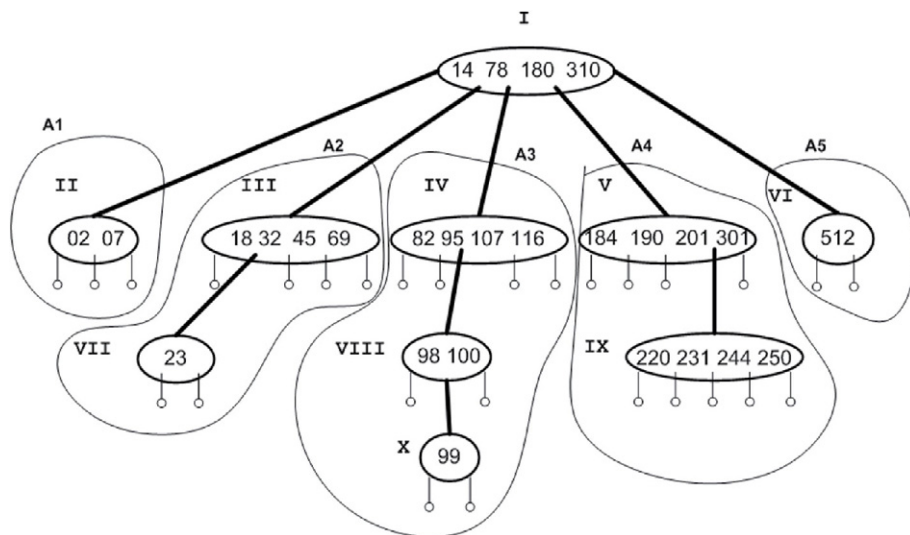


FIGURA 12.1: Árvore N -ária AN.

Para ilustrar essa definição, vamos considerar a árvore AN em que temos a raiz I e as subárvores A1, A2, A3, A4 e A5 (observe a [Figura 12.1](#)).

Observe que em AN, conforme descrito, cada nó pode conter determinado número de chaves (elementos), diferentemente de uma árvore binária, em que, em cada nó, existe uma única chave (elemento).

Conceitos

Numa árvore N -ária é fácil perceber a relação entre o número de nós filhos que a árvore suporta (subárvores de um nó) e o número de chaves (elementos) dentro do nó. No exemplo da árvore AN ([Figura 12.1](#)), temos uma árvore em que cada nó pode ter até cinco subárvores filhas e, consequentemente, até quatro chaves (elementos) por nó.

Outra observação importante refere-se ao fato de termos em AN nós que contêm um número de chaves inferior à capacidade máxima do nó – no caso nós II, VI, VII, VIII e X. Nós como esses são chamados de *nós incompletos*. De outro lado, temos nós cujas capacidades máximas de chaves foram atingidas (nós I, III, IV, V e IX), que são chamados de *nós completos*.

Para armazenar as chaves de um nó, em geral utilizamos um vetor, dada a simplicidade de sua manipulação e economia de espaço possíveis quando se evitam os apontadores dinâmicos para as chaves seguintes contidas no nó.

De outro lado, pelo fato de existirem nós incompletos, o uso do vetor leva ao desperdício de espaço de armazenamento dentro do nó, uma vez que, mesmo não havendo um número de chaves máximo, temos o espaço no vetor alocado.

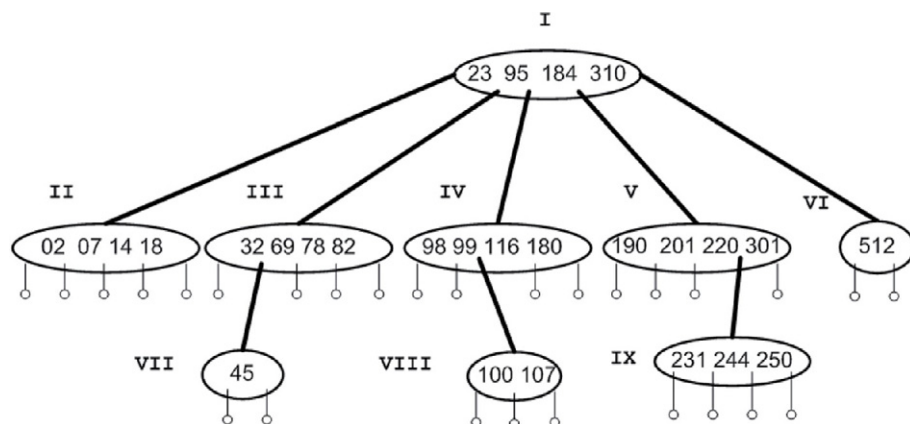


FIGURA 12.2: Árvore N -ária *topdown*.

Assim, é interessante evitarmos a existência de nós incompletos, procurando sempre distribuir as chaves de forma que o maior número possível de nós existentes seja completo.

Além disso, a manutenção de nós completos viabiliza a redução do número de nós (com conseqüente redução na altura da árvore), o que, em caso de busca por determinada chave, leva à redução do número de acessos aos nós da árvore. Dessa forma, podemos considerar que nas árvores N -árias é importante tentar manter o maior número de nós completos possível.

Nesse contexto, outro conceito importante refere-se à chamada árvore N -ária *topdown* (de cima para baixo). A existência dessa árvore se dá a partir da satisfação da condição de que qualquer nó incompleto deve necessariamente ser um nó folha.

Para ilustrar melhor o conceito de árvore N -ária *topdown*, observe a [Figura 12.2](#).

Observe que, na árvore da [Figura 12.2](#), todos os nós que não são folhas (I, III, IV e V) são nós completos.

Ampliando os conceitos, tomemos o exemplo da [Figura 12.3](#). Observe que a árvore nele representada tem tal distribuição de chaves pelos nós que fazem seus nós folhas (VII, VIII, IX, X e XI) estarem no mesmo nível. Essa árvore é considerada *balanceada* (pela altura da árvore) e será tópico de discussão no Capítulo 13. Por enquanto, pense nas possíveis vantagens e desvantagens que ela poderia trazer.

A árvore da [Figura 12.3](#), embora balanceada, não é uma árvore N -ária *topdown*, pois apresenta nós não folhas que são incompletos. Para compreender melhor esse conceito, construa uma árvore N -ária *topdown* balanceada.

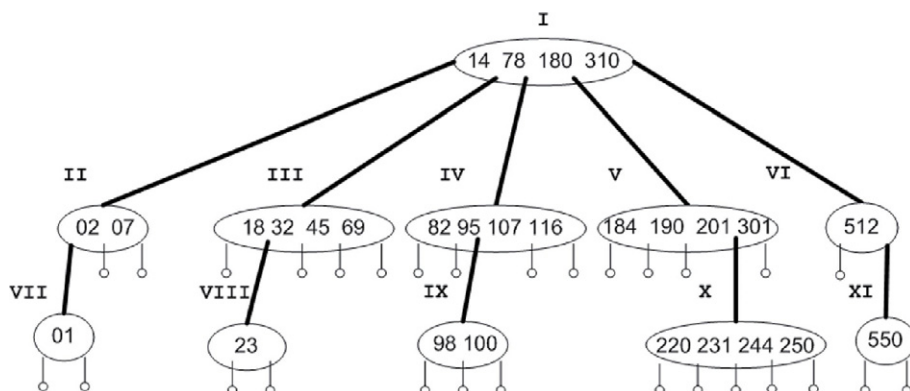


FIGURA 12.3: Árvore N -ária balanceada.



Conceito

O número máximo de chaves no nível N em uma árvore N -ária será o produto de número máximo de chaves do nível $N-1$ pelo grau da árvore. Dessa forma, uma árvore N -ária de grau 5, no nível 0 terá até 4 chaves, no nível 1 terá até 20 chaves, no nível 2 terá até 100 chaves, e assim por diante.

Implementação de uma árvore N -ária

Numa árvore N -ária, devido à variação que um nó dessa árvore pode apresentar em termos de número de chaves e subárvores, a forma comumente adotada para sua implementação é aquela voltada ao uso de alocação dinâmica de memória.

Embora não exista uma forma única de fazer essa implementação, neste livro optamos por representar um nó da árvore N -ária por meio da definição de um registro contendo um valor numérico, em que é armazenado o número de chaves existentes no nó (o máximo será N); uma lista de N posições que armazenará as chaves (elementos) contidas no nó; e uma lista de $N+1$ posições que armazenará os apontadores para as subárvores desse nó.

Dica



Para armazenar os dados secundários dos elementos que farão parte do nó da árvore binária, bastaria alterar os elementos para conter não apenas o campo chave, mas também campos adicionais em que os dados secundários ficariam associados aos respectivos elementos. Porém, por motivo de simplificação, como explicamos no início deste capítulo, ficaremos restritos ao armazenamento das chaves como representantes do elemento armazenado.

Na [Figura 12.4](#), temos a representação de um nó dessa árvore:



FIGURA 12.4: Representação de um nó da árvore N -ária.

Declaração de uma árvore N -ária

Consiste na definição do nó representado na [Figura 12.4](#) através de um registro. Para representar os campos descritos na representação desse nó, temos: *nro_chaves* (armazena o número de chaves que estão armazenadas no nó, em determinado momento), *chaves* (vetor com N posições que armazena as chaves), e *apontadores* (vetor com $N+1$ posições que armazena os apontadores para as subárvores).

Código 12.1

```
typedef struct tipo_no no;
struct tipo_no
{
    int nro_chaves;
    tipo_dado chaves[N];
    tipo_no *apontadores[N+1];
};
```

Assim como ocorre nas árvores binárias, nos exemplos que serão apresentados sobre árvores N -árias vamos adotar que *tipo_dado* será um valor do tipo *inteiro*. Assim:

```
typedef int tipo_dado;
```

Operações básicas numa árvore N -ária

A manipulação de uma árvore N -ária pode ocorrer por diferentes operações. Assim como fizemos com as árvores binárias, vamos focar as operações básicas de maior utilidade e difusão: *percurso* (*busca*), *inserção* e *remoção*.

Percurso de árvore N -ária

Na definição de árvores N -árias (de busca) que estamos abordando neste capítulo, temos que, à esquerda de cada elemento E_i de um nó da árvore, no caso de existir uma subárvore, ela conterá elementos com chaves menores que a chave de E_i ; por conseguinte, à direita de cada elemento E_i de um nó, no caso de existir uma subárvore, ela conterá elementos com chaves maiores que a chave de E_i .

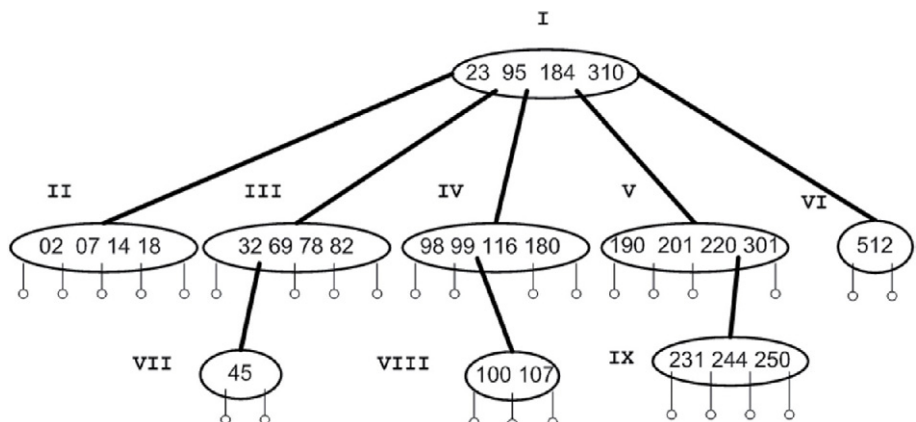
Assim, é desejável a existência de uma forma de percurso que permita a apresentação das chaves de forma ordenada crescente, que aqui denominaremos *percurso ordenado*.

Percurso ordenado (apresentação das chaves em ordem crescente)

Nessa forma de percurso, considerando-se o registro apresentado na [Figura 12.4](#), temos (se raiz for um nó não vazio):

- para o nó raiz, obtém-se no campo *nro_chaves* o número de chaves N_i presentes no nó;
- inicia-se um processo repetitivo, em que um índice j é inicializado com 0 e vai variar até $N_i - 1$ (de 0 até $N_i - 1$ temos N_i chaves, portanto, em $N_i - 1$ temos a última chave). Para cada j desse intervalo, percorre-se em percurso ordenado a j -ésima subárvore apontada no nó raiz e, na sequência, apresenta na saída a j -ésima chave do nó raiz;
- percorre em percurso ordenado a última subárvore apontada no nó raiz.

Reproduzindo novamente a árvore da [Figura 12.2](#), vejamos como ficaria a ordem de apresentação de seus elementos num percurso ordenado, bem como a sequência dos nós visitados.



PERCURSO ORDENADO: 02-07-14-18-23-32-45-69-78-82-95-98-99-100-107-
116-180-184-190-201-220-231-244-250-301-310-512

Nós visitados: I-II-I-III-VII-III-I-IV-VIII-IV-I-V-IX-V-I-VI

A seguir, tem-se a implementação da função *ordenado*, que apresenta como saída os elementos da árvore *N*-ária, segundo os critérios já descritos.

Código 12.2

```
void ordenado(no *Raiz)
{
    int j, Ni;

    if (Raiz != NULL)
    {
        Ni = Raiz->nro_chaves;           // Obtém o número de chaves Ni do nó i
        for (j = 0; j < Ni; j++)        // Percorre as Ni chaves do nó i
        {
            ordenado(Raiz->apontadores[j]); // Percorre a j-ésima subárvore do nó i
            printf("%d ", Raiz->chaves[j]); // Apresenta a j-ésima chave do nó i
        }
        ordenado(Raiz->apontadores[j]); //Percorre a última subárvore do nó i
    }
}
```

Busca por um elemento (por meio de sua chave)

Outra operação básica importante refere-se à busca de determinada chave associada a um elemento da árvore.

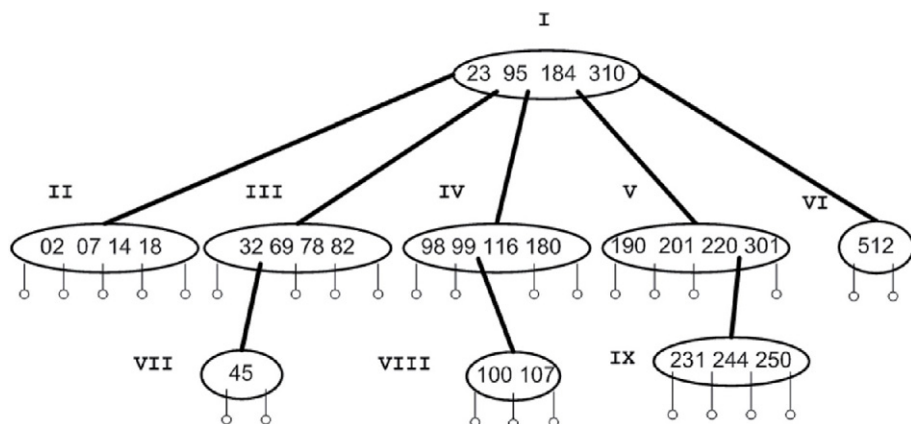
Para realizar essa tarefa, inicialmente vamos considerar que o número máximo de elementos dentro de um nó é reduzido a ponto de ser mais interessante percorrer essa lista interna ao nó de forma sequencial do que usando outra abordagem qualquer de busca em listas.

Dessa forma, considerando-se o registro apresentado na [Figura 12.4](#), temos que a ideia geral dessa BUSCA para determinada chave Ch, usando recursividade, seria:

- verifica se raiz é nó vazio (NULL), então retorna o endereço NULL (como apontador do nó) e a posição -1, sinalizando que a chave não foi encontrada;
- caso contrário, inicializa-se *i* com o endereço do nó raiz. Para o nó *i*, no campo *nro_chaves* obtém-se o número de chaves *Ni* presentes no nó e coloca-se em prática os passos seguintes;

- inicia-se um processo repetitivo em que um índice j é inicializado com 0 e varia até N_i-1 (de 0 até N_i-1 temos N_i chaves, portanto, em N_i-1 temos a última chave);
- dentro do processo repetitivo: verifica se ($Ch = j$ -ésima chave), então finaliza o processo de BUSCA (j já estará com a posição da chave procurada dentro do nó i). Caso contrário, verifica se ($Ch < j$ -ésima chave), então interrompe o processo repetitivo, tendo em j a posição do vetor apontadores do nó i , que aponta a subárvore onde pode estar Ch ;
- Se o processo repetitivo chegou até o final é porque ou $Ch < j$ -ésima chave ou Ch é maior que a maior chave armazenada no nó i . Nas duas situações, temos em j a posição do vetor apontadores do nó i , que contém o endereço da subárvore do nó i , por onde será reiniciado o processo de BUSCA pela posição de inserção de Ch .

Reproduzindo novamente a árvore da [Figura 12.2](#), vejamos como ficaria a ordem das chaves e dos nós visitados na busca pela chave 100, bem como os valores de i e j retornados.



Ordem de chaves visitadas: 23-95-184-98-99-116-100

Ordem dos nós visitados: I-IV-VIII

Retorno: i com o endereço do nó VIII

j com a posição 0

Agora, vejamos como ficaria na busca pela chave 203 (inexistente).

Ordem de chaves visitadas: 23-95-184-310-190-201-220

Ordem dos nós visitados: I-V

Retorno: *i* com o endereço NULL

j com a posição -1

Para fixar, mostre como ficaria, nas buscas pelas chaves 250 e 181, as ordens de chaves e dos nós visitados, bem como os valores de *i* e *j* retornados.

Em seguida, tem-se a implementação da função *busca_no*, que buscará pela chave *Ch* na árvore raiz, retornando em *i* o endereço do nó onde está a chave *Ch* e em *j* a posição dessa chave dentro desse nó. No caso de inexistência da chave *Ch*, retornará *i* como NULL e *j* como -1, conforme já descrito.

Código 12.3

```
void busca_no(no *Raiz, tipo_dado Ch, no **i, int *j)
{
    int Ni;
    if (Raiz == NULL)           // Não existe mais subárvores para procurar Ch, ou seja, Ch não existe
    {
        *i = NULL;
        *j = -1;
    }
    else
    {
        *i = Raiz;
        Ni = (*i)->nro_chaves;
        for (*j = 0; *j < Ni; (*j)++)      // Primeira posição do vetor chaves é 0(zero)
        {
            printf("Chave visitada= %d\n", (*i)->chaves[*j]); //Mostra a chave visitada durante a busca
            if ((*i)->chaves[*j] == Ch)      // Encontrou a chave Ch procurada
            {
                return;                      // Finaliza a busca, sendo i o nó e j a posição da chave Ch
            }
            else if (Ch < (*i)->chaves[*j]) // Chave Ch deve estar na j-ésima subárvore apontada no nó i
                break;                      // Interrompe o laço para buscar a chave Ch na j-ésima subárvore
        }
        busca_no((*i)->apontadores[*j], Ch, &(*i), &(*j));
    }
}
```

Uma questão que pode ser levantada é a situação em que o número de chaves em cada nó se torna tão elevado que a substituição da busca sequencial pelas chaves dentro de um nó por outro mecanismo de busca pode trazer algum benefício. Assim, para ampliar seu conhecimento, tente alterar a função *busca_no* para que a busca pelas chaves dentro de um nó use um dos mecanismos vistos no Capítulo 4 que apresente melhor desempenho no caso de grande volume de dados em uma lista ordenada.

Inserção de um elemento (chave) numa árvore N-ária

Quando abordamos a inserção numa árvore *N*-ária (de busca), adotamos uma ideia similar à usada nas árvores binárias, vistas no Capítulo 11, no que se refere à distribuição das chaves de forma ordenada. Essa distribuição foi descrita novamente na definição de árvores *N*-árias dada no início deste tópico. Além disso, numa árvore *N*-ária, para explorarmos o benefício de poder armazenar várias chaves num nó, a inserção sempre priorizará o preenchimento de nós nos níveis mais inferiores possíveis da árvore.



Atenção

Adotaremos que na árvore *N*-ária não será admitida a possibilidade de existência de chaves duplicadas.

A partir dessas considerações, temos que a inserção se dará em duas etapas básicas: *localização* do nó e posição dentro dele para a inserção da chave, e *inserção com possível rearranjo* das chaves e subárvores desse nó para manutenção da ordenação.

Localização do ponto para inserção da chave (elemento)

Na localização do nó e posição para inserção da chave, usaremos uma variação da função *busca_no* vista, a que chamaremos *busca_no_ins*. A

diferença é que *busca_no_ins* não procura pela chave, e sim pela posição na árvore para sua inserção. No caso de existência da chave a ser inserida, devolve um alerta, evitando sua duplicidade. Assim, *busca_no_ins* contém os seguintes passos:

- se raiz é nó vazio (NULL) é porque será necessário criar um novo nó para inserção da chave;
- caso contrário, inicializa-se *i* com o endereço do nó raiz. Para o nó *i*, obtém-se, no campo *nro_chaves*, o número de chaves *Ni* presentes no nó;
- inicia um processo repetitivo, em que um índice *j* é inicializado com 0 e vai variar até *Ni-1* (de 0 até *Ni-1* temos *Ni* chaves, portanto, em *Ni-1* temos a última chave);
- dentro do processo repetitivo: verifica se ($Ch = j$ -ésima chave), então retorna a posição -1 para *j*, sinalizando que a chave já existe na árvore. Caso contrário, verifica se ($Ch < j$ -ésima chave), então se existe espaço para a chave *Ch* no nó *i* e interrompe o processo de BUSCA, tendo em *j* a posição do vetor chaves do nó *i*, onde deverá ser inserida a chave *Ch*. Caso não exista espaço, interrompe o processo repetitivo, tendo em *j* a posição do vetor apontadores do nó *i* que aponta a subárvore em que será inserida *Ch*;
- se o processo repetitivo chegou até o final é porque ou $Ch < j$ -ésima chave e não existe espaço para inserção da *Ch* no nó *i*, ou *Ch* é maior que a maior chave armazenada no nó *i* e não existe espaço para sua inserção nesse nó. Nas duas situações, temos em *j* a posição do vetor apontadores do nó *i*, que contém o endereço da subárvore do nó *i* por onde será reiniciado o processo de BUSCA pela posição de inserção de *Ch*.

A seguir, tem-se a implementação da função *busca_no_ins*, que buscará a posição de inserção da chave *Ch*, conforme já descrito. Note que GRAU será uma constante contendo o grau da árvore *N*-ária.

Código 12.4

```
void busca_no_ins(no *Raiz, tipo_dado Ch, no **i, int *j)
{
    int Ni;

    // Se (Raiz == NULL) é porque não existe mais subárvores para procurar Ch e o nó i está sem espaço.
    // Finaliza a busca, já tendo a posição j do nó i como ponto onde o novo nó deverá ser criado para inserção de Ch
    if (Raiz != NULL)
    {
        *i = Raiz;
        Ni = (*i)->nro_chaves;
        for (*j = 0; *j < Ni; (*j)++) // Procura pela 1ª chave do nó i maior que Ch
        {
            if ((*i)->chaves[*j] == Ch) // Chave Ch já existe na árvore
            {
                *j = -1;
                return; // Finaliza a BUSCA
            }
            else if (Ch < (*i)->chaves[*j]) // Encontrou no nó (*i) uma chave maior que Ch
            {
                if (Ni < GRAU-1) // Existe espaço no nó i para inserção de uma chave Ch menor que uma já existente
                    return; // Finaliza a busca, sendo i o nó para inserção e j a posição da chave
                else
                    break; // Interrompe o laço para buscar a chave Ch na j-ésima subárvore
            }
        }
        busca_no_ins((*i)->apontadores[*j], Ch, &(*i), &(*j));
    }
}
```

Inserção da chave (elemento) na árvore *N*-ária

Para a inserção de uma chave, devem ser verificadas as seguintes etapas:

- verificar se Ch será a primeira chave da árvore. Sendo a primeira chave, deve ser criado e inicializado um novo nó, que passará a ser a raiz da árvore;
- caso contrário, inicializa-se *i* com o endereço do nó raiz e *j* como posição 0 dentro do nó;
- chama pela função *busca_no_ins*, que devolverá se Ch já existe ou se os valores de *i* e *j* orientarão a inserção;

- verifica se não existe mais espaço no nó i para a inserção de Ch , então cria, inicializa e insere Ch como a primeira chave de um novo nó E , associando-o com a j -ésima subárvore do nó i ;
- caso contrário, verifica se Ch será a última chave do nó i , então cria uma subárvore vazia na $(j+1)$ -ésima posição;
- caso contrário, rearranja o nó i , mantendo a ordenação de suas chaves e subárvores. Para isso, desloca todas as chaves e subárvores (da j -ésima até a última posição preenchida nesse nó) de uma posição para a direita e cria uma subárvore vazia na j -ésima posição;
- por fim, para completar as etapas 5 e 6, insere Ch na j -ésima posição e incrementa o número de chaves do nó.

A seguir, tem-se a implementação da função *inserir_n_naria*, que fará a inserção da chave Ch , conforme já descrito. Note que GRAU será uma constante contendo o grau da árvore N -ária.

Código 12.5

```

void inserir_n_aria (no **Raiz, tipo_dado Ch)
{
    no *i; // Apontador para o nó onde será inserida a chave Ch
    int j; // Posição no nó para inserção ordenada da chave Ch
    int Ni, k;

    if (*Raiz == NULL) // Primeiro elemento da árvore
    {
        *Raiz = (no *)malloc(sizeof(no)); // Aloca espaço na memória correspondente ao nó Raiz
        for (k = 0; k < GRAU-1; k++)
        { // Inicializa o nó com vazio
            (*Raiz)->chaves[k] = -1;
            (*Raiz)->apontadores[k] = NULL;
        }
        (*Raiz)->apontadores[k] = NULL; // Inicializa o endereço da última subárvore
        (*Raiz)->chaves[0] = Ch; // Insere o conteúdo (chave) como primeira chave do nó E
        (*Raiz)->nro_chaves = 1; // Inicializa o número de chaves contidas no nó
    }
    else
    {
        i = *Raiz; // Inicializa i e j
        j = 0;
        busca_no_ins (*Raiz, Ch, &i, &j);
        if (j == -1)
            printf("Chave já existente na árvore\n");
        else
        {
            Ni = i->nro_chaves;
            if (Ni == GRAU-1) // Não existe espaço no nó para inserção de Ch
            { // Cria novo nó E
                no *E;
                E = (no *)malloc(sizeof(no)); // Aloca espaço na memória correspondente ao nó E
                for (k = 0; k < Ni; k++)
                { // Inicializa o nó com vazio
                    E->chaves[k] = -1;
                    E->apontadores[k] = NULL;
                }
                E->apontadores[k] = NULL; // Inicializa o endereço da última subárvore
                E->chaves[0] = Ch; // Insere o conteúdo (chave) como primeira chave do nó E
                E->nro_chaves = 1; // Inicializa o número de chaves contidas no nó
                i->apontadores[j] = E; // Insere o nó E como a j-ésima subárvore do nó i
            }
            else
            {
                if (j == Ni) // A chave Ch será a última do nó i
                    i->apontadores[j+1] = NULL; // Inicializa o endereço da última (jésima+1) subárvore
                else
                { // Rearranja o nó i, mantendo a ordenação de suas chaves e subárvores
                    i->apontadores[Ni+1] = i->apontadores[Ni]; // Desloca última subárvore
                    for (k = Ni; k > j; k--) // Abre espaço para a chave Ch, deslocando as chaves e subárvores
                    {
                        i->chaves[k] = i->chaves[k-1];
                        i->apontadores[k] = i->apontadores[k-1];
                    }
                    i->apontadores[j] = NULL; // Inicializa o endereço da j-ésima subárvore
                }
                i->chaves[j] = Ch; // Insere a chave Ch na posição j
                i->nro_chaves = i->nro_chaves + 1; // Incrementa o número de chaves contidas no nó i
            }
        }
    }
}

```

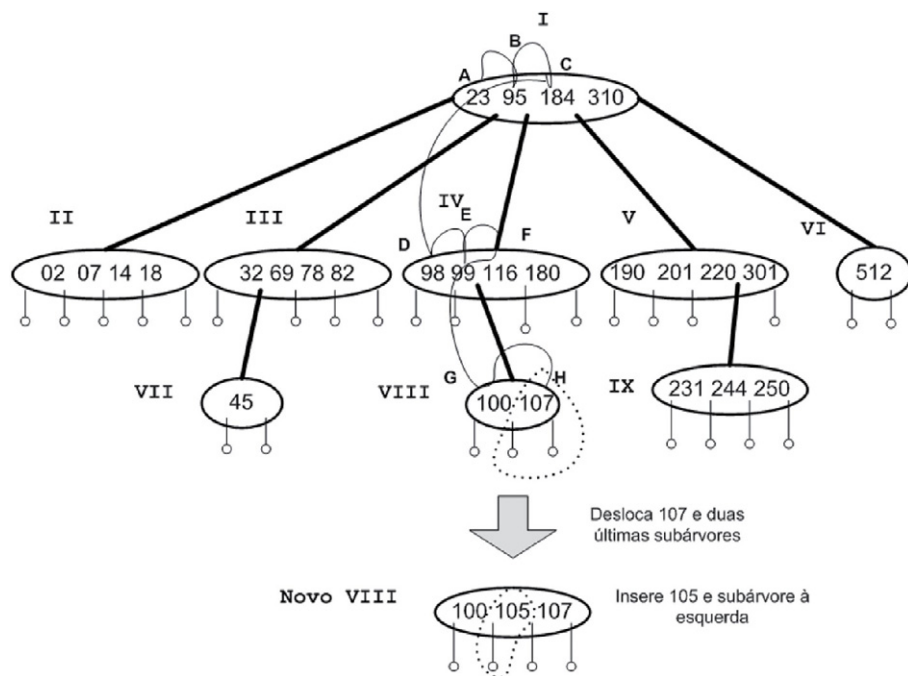


FIGURA 12.5: Inserção da chave 105.

Para ilustrar como será a inserção de chaves numa árvore N -ária, vejamos como ficaria a inserção da chave 105 na árvore da [Figura 12.5](#).

Visita: nó I, chaves (A)23-(B)95-(C)184

não existe espaço no nó I

Visita: nó IV, chaves (D)98-(E)99-(F)116

não existe espaço no nó IV

Visita: nó VIII, chaves (G)100-(H)107

existe espaço no nó VIII

desloca subárvores e chaves para abrir espaço para 105

insere 105

cria subárvore vazia à esquerda de 105

Para fixar, mostre como ficaria a inserção das chaves 01 e 513.

É importante notar que a forma de inserção apresentada não considera a possibilidade de existirem subárvores não vazias apontadas por nós que não estejam completos (máximo de chaves). Além disso, não existe uma preocupação em redistribuir as chaves pelos nós, de forma a garantir que um novo nível na árvore somente seja iniciado quando todos os nós dos níveis existentes forem completos (tenham atingido o número máximo de chaves).

Remoção de um elemento (*chave*) numa árvore N -ária

Ao pensarmos em simplicidade na remoção de elementos numa estrutura de dados qualquer, em geral, o que se adota é não remover de fato a chave, mas apenas sinalizar de alguma forma (marca) que naquela posição aquela chave não existe mais. Essa abordagem oferece maior facilidade na operacionalização da remoção, mas provoca desperdício de espaço de armazenamento.

No caso das árvores N -árias, além do desperdício de espaço mencionado, temos dificuldade em garantir a manutenção da ordenação com tal ação. Isso tudo leva à ideia de reaproveitar o espaço deixado numa futura reinserção da mesma chave. Para chaves diferentes da removida, não há como garantir a posição em relação às *chaves anterior e posterior* na árvore, o que obrigaria sua reorganização. Como *chave anterior* consideramos aquela que possui valor imediatamente menor do que a chave em questão e, de forma análoga, como *chave posterior*, aquela com valor imediatamente maior. Por exemplo, numa lista de chaves com 4-9-12-34-47 para a chave 12, temos 9 como sua chave anterior e 34 como sua chave posterior.

Para melhor compreensão, vejamos como ficaria a remoção da chave 95 presente na árvore da [Figura 12.6](#).

Note que, ao remover 95, deixando em seu lugar uma marca, toda a ordenação e a estruturação da árvore ficarão mantidas, porém com o desperdício daquele espaço de armazenamento.

Se for necessária a reinserção da chave 95, basta armazená-la no seu lugar original (veja a [Figura 12.7a](#)). De outro lado, imagine que desejamos inserir a chave 99. Seria possível simplesmente reaproveitar o espaço marcado? A resposta é não, porque perderíamos a ordenação entre as chaves (veja a [Figura 12.7b](#)).

Apesar de simples, o reaproveitamento, sendo limitado a chaves iguais (removida e inserida), apenas se mostra adequado quando a situação-problema apresenta com frequência esse tipo de ocorrência, o que não é comum.

Isso nos leva à necessidade de buscar alternativas para a remoção de chaves em árvores N -árias. Nesse contexto, nos deparamos com a possibilidade de repetir a ideia de remoção vista para as árvores binárias. Assim, temos as seguintes situações para a chave a ser removida:*

1. **quando não apresenta subárvore à esquerda:** basta rearranjar (compactar) o registro do nó, deslocando todas as chaves e subárvores

*Considerando-se o registro da [Figura 12.4](#), o que aqui chamaremos de *subárvore à esquerda* da j -ésima chave de um nó i , seria o que lá consideramos a j -ésima subárvore apontada por i , por conseguinte, a *subárvore à direita* dessa chave seria a $(j+1)$ -ésima subárvore apontada por i .

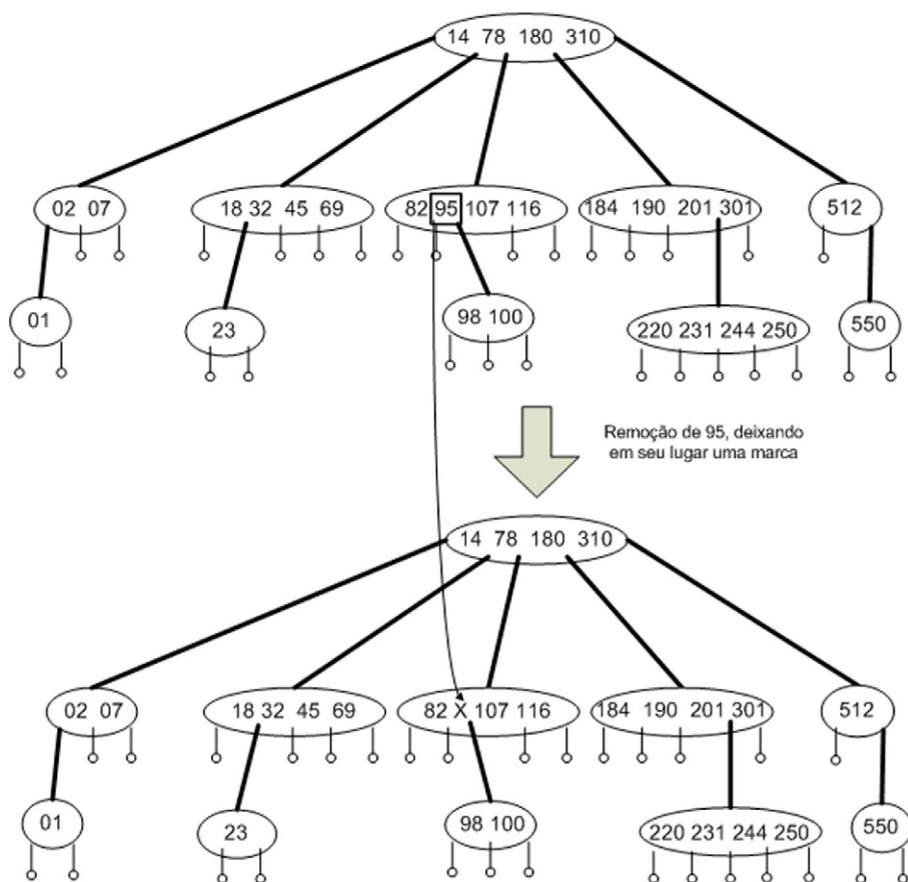
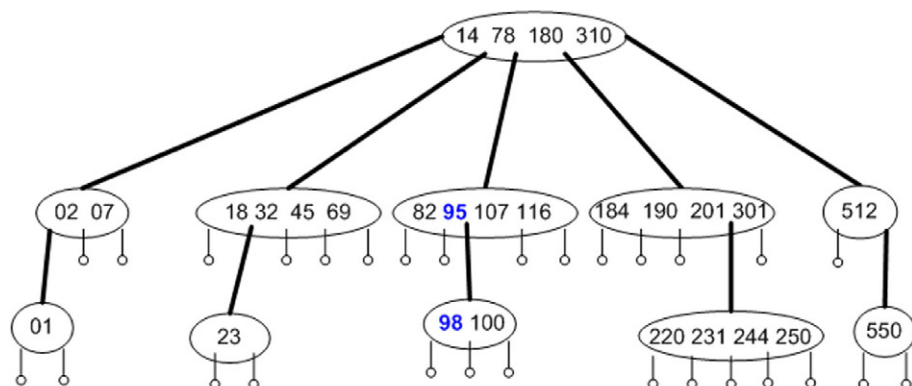


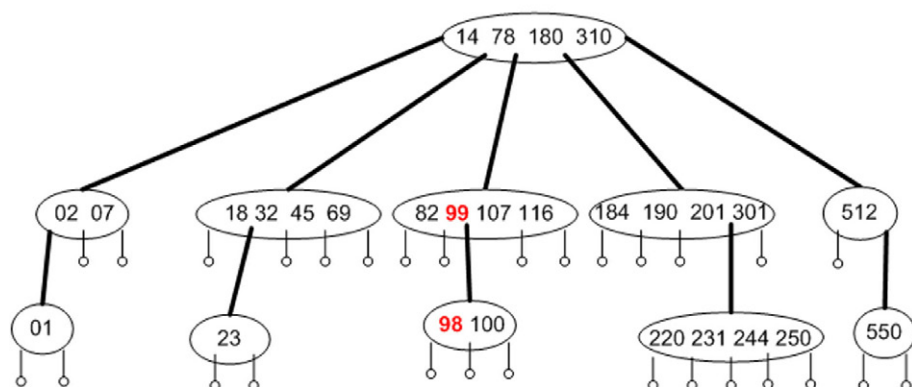
FIGURA 12.6: Remoção colocando uma marca na chave removida.

(da posição posterior no nó até a última existente nesse nó) de uma posição para a esquerda, diminuindo em um o número de chaves no nó. Se a chave for a última do nó, basta deslocar a subárvore à direita de uma posição para a esquerda, diminuindo em um o número de chaves no nó;

2. **quando não apresenta subárvore à direita:** basta rearranjar (compactar) o registro do nó, deslocando todas as chaves (da posição posterior no nó até a última existente nesse nó) de uma posição para a esquerda e deslocando todas as subárvores (da subárvore à direita da chave posterior no nó até a última apontada pelo nó) também de uma posição para a esquerda, diminuindo em um o número de chaves no nó. Se a chave for a última do nó, neste caso, basta diminuir em um o número de chaves do nó;



(a)



(b)

FIGURA 12.7: Reinserção da chave 95 e inserção da chave 99.

3. **quando apresenta subárvores à esquerda e à direita:** deve ser deslocada para a posição em que se encontra a chave removida, sua chave anterior e repetir as etapas 1, 2 e 3, tendo em mente a remoção da chave deslocada do nó em que ela estava originalmente. Esse processo se repete enquanto há necessidade de deslocar chaves anteriores[†]
4. Em todos os casos, quando a chave removida (ou última deslocada) é única no nó, deve-se eliminá-lo da árvore.

[†] Assim como ocorre nas árvores binárias, a escolha da chave a ser deslocada poderia ser pela chave posterior em vez da anterior.

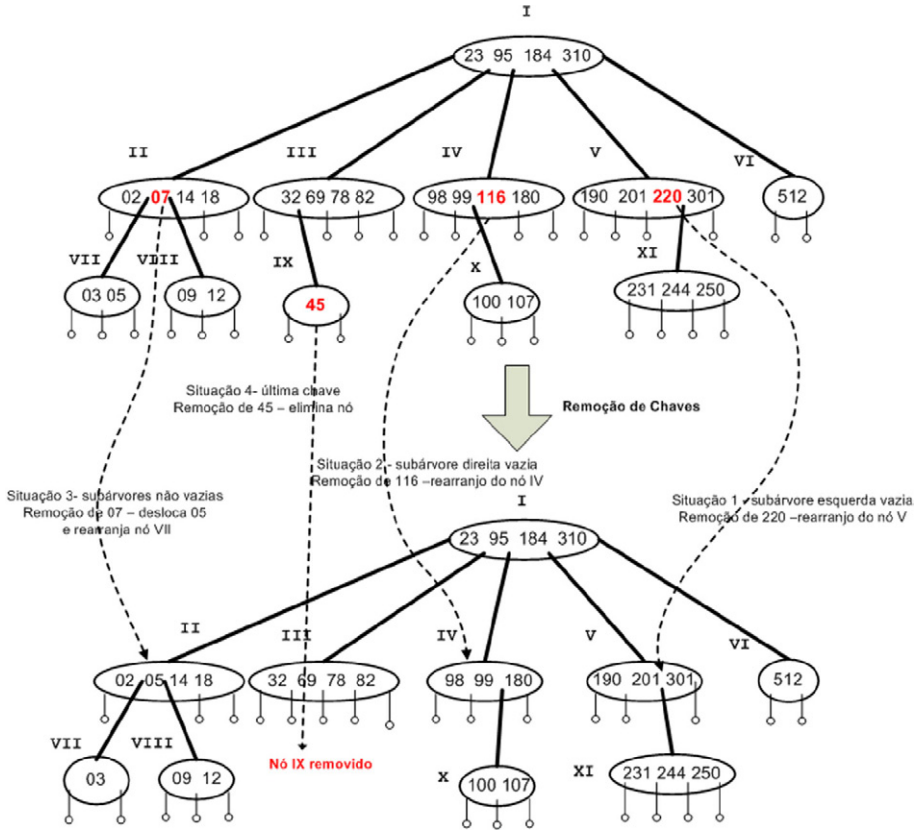


FIGURA 12.8: Remoções de chaves numa árvore N -ária.

Para ilustrar, vamos observar um exemplo para cada situação apresentada (veja a Figura 12.8), em que:

- para a situação 1, considere a remoção da chave 220. Essa chave não possui subárvore à esquerda, assim, o nó V é compactado deslocando-se as chaves e subárvores à direita de 220 de uma posição à esquerda, nas listas de chaves e apontadores para subárvore desse nó;
- para a situação 2, considere a remoção da chave 116. Essa chave não possui subárvore à direita, assim, o nó IV é compactado eliminando-se o apontador da subárvore vazia à direita de 116 e deslocando-se as chaves e subárvores restantes à direita de 116 de uma posição à esquerda, nas listas de chaves e apontadores para subárvore desse nó;

- para a situação 3, considere a remoção da chave 07. Essa chave possui subárvores à esquerda e à direita. Nessa situação, a chave 07 é substituída pela maior chave (05) existente na subárvore à esquerda, e o nó V, em que estava a chave deslocada, é compactado;
- para a situação 4, considere a remoção da chave 45. Nessa situação, 45 é a única chave do nó IX, que, portanto, deve ser eliminado.

No caso de necessidade de estudos mais aprofundados, procure elaborar funções que contemplem os algoritmos apresentados para a remoção de chaves numa árvore *N*-ária, tanto por meio da colocação de marcas, como da remoção com rearranjo dos nós. Lembre-se de verificar a necessidade de eventuais ajustes na função de inserção, anteriormente apresentada, para seu correto funcionamento em conjunto com suas implementações.

Vantagens e uma aplicação típica de árvores *N*-árias

De modo geral, foi possível observar que, ao permitir que sejam colocadas várias chaves num único nó, temos a possibilidade de reduzir de forma significativa o número de nós necessários e a altura da árvore. Assim, espera-se menor número de acessos a diferentes nós quando se deseja localizar determinada chave (elemento) dentro da estrutura de árvore *N*-ária, se comparada à árvore binária.

Com isso, podemos considerar que quanto maior o volume de dados a ser armazenado numa estrutura de árvore, maior será a vantagem em utilizar uma árvore *N*-ária. Porém, não podemos perder de vista que sua manipulação apresenta maior necessidade de reorganização da estrutura de árvore existente, o que aumenta a complexidade de implementação e também o tempo despendido com a reorganização da árvore.

Por causa dessas características, esse tipo de árvore se mostra adequada para uso em situações nas quais os dados são armazenados em dispositivos externos (por exemplo, disco). Esses dispositivos são demasiado lentos, se comparados com a memória interna do computador. Isso decorre principalmente do tempo gasto para o correto posicionamento do dispositivo para a leitura dos dados. De outro lado, quando é feito o acesso a esses dispositivos, a leitura sequencial dos dados subsequentes, a partir do posicionamento, não é demorada. Assim, considerando-se que a cada leitura seria possível ler todos os elementos de um nó da árvore, teríamos um ganho significativo de desempenho com o uso desse tipo de estrutura, além da forte redução do espaço de busca que ele oferece. Essa redução decorre do fato de que, ao se definir a subárvore seguinte a ser investigada, todas as restantes (que carregam um número considerável de

elementos) já são eliminadas do espaço de busca restante. Apenas para exemplificar, enquanto numa árvore binária espera-se eliminar em média 50% do espaço de busca a cada nova subárvore escolhida, numa árvore N -ária, por exemplo de grau 5, a redução média pode chegar a 80% do espaço de busca.

Eficiência na busca numa árvore N -ária

Da mesma forma como visto para as árvores binárias no Capítulo 11, as operações básicas numa árvore N -ária tem tempo proporcional à sua altura. A altura da árvore, nesse caso, dependerá da ordem de inserção das chaves, do número N total de chaves e da ordem m da árvore.

Assim, ao considerarmos a situação ideal de uma árvore N -ária, em que todos os nós estão preenchidos com o número máximo de chaves possível e todos os nós folhas estão no mesmo nível, teremos, no pior caso, a execução das operações num tempo $O(\log_m N)^\ddagger$.

Para termos uma comparação superficial com as árvores binárias (considerando tanto a árvore N -ária quanto a binária com a melhor situação de distribuição de nós), teríamos, de forma aproximada, o seguinte tempo de execução:

- Com $N = 1.000$ e $m = 10$

Árvore binária $\approx O(\log_2 1.000) \approx O(9,97)$

Árvore N -ária $\approx O(\log_{10} 1.000) \approx O(3)$

- Com $N = 10.000.000$ e $m = 10$.

Árvore binária $\approx O(\log_2 10.000.000) \approx O(23,25)$

Árvore N -ária $\approx O(\log_{10} 10.000.000) \approx O(7)$

Esses exemplos ilustram o potencial de maior eficiência das árvores N -árias em termos de tempo de execução das operações básicas, principalmente quando se considera um grande volume de dados. Para aprofundamento sobre a eficiência de uma árvore N -ária, recomendamos a leitura do material mencionado na seção “Para Saber Mais” deste capítulo.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, nas próximas seções veremos implementações nas linguagens de programação Java e Phython.

‡ Em termos de eficiência em algoritmos, a notação $O(\dots)$ representa a ordem associada ao que se deseja mensurar. Assim, $O(\log)$ representa uma ordem logarítmica.

Java

Código 12.1

```
public class NoNaria {

    private int nroChaves;
    public int[] chave;
    public NoNaria[] nosFilhos;
    private int grau;

    public NoNaria(int grau) {
        nroChaves = 0;
        nosFilhos = new NoNaria[grau];
        chave = new int[grau-1];
        this.grau = grau;
    }

    public boolean isFull () {
        return nroChaves == this.grau-1;
    }

    public int getNroChaves () {
        return this.nroChaves;
    }

    public void adicionaValor (int valor) {
        this.chave[nroChaves] = valor;
        this.nroChaves++;
    }
}

public class ArvoreNaria {

    NoNaria raiz;
    private int grau;

    public ArvoreNaria(int grau) {
        this.raiz = null;
        this.grau = grau;
    }
}
```

Código 12.2

```
public void ordenado() {
    ordenado(this.raiz);
}

private void ordenado(NoNaria no) {
    if(no != null) {
        int j = 0;
        for (j = 0; j < no.getNroChaves(); j++) {
            ordenado(no.nosFilhos[j]);
            System.out.print(no.chave[j] + " ");
        }
        ordenado(no.nosFilhos[j]);
    }
}
```

Código 12.3 e Código 12.4

```
public NoNaria busca_no(int valor) {
    if (this.raiz == null) {
        return null; // Nao existe o valor
    } else {
        return busca_no(this.raiz, valor);
    }
}

private NoNaria busca_no(NoNaria no, int valor) {
    if (no == null) {
        return null;
    } else {
        for (int i = 0; i < no.getNroChaves(); i++) {
            if (valor == no.chave[i]) {
                return no; // Valor duplicado
            }
            if (valor < no.chave[i]) {
                return busca_no(no.nosFilhos[i], valor);
            }
        }
        if (no.getNroChaves() == grau - 1) {
            if (valor > no.chave[grau - 2]) {
                return busca_no(no.nosFilhos[grau - 1], valor);
            } else {
                return null;
            }
        } else {
            return null;
        }
    }
}
```

Código 12.5

```
public void inserir_n_naria(int valor) {
    if (this.raiz == null) {
        this.raiz = new NoNaria(this.grau);
        this.raiz.adicionaValor(valor);
    } else {
        adicionaReg(this.raiz, valor, raiz, 0);
    }
}

private void adicionaReg(NoNaria no, int valor, NoNaria pai, int indice) {
    if (no == null) {
        no = new NoNaria(this.grau);
        no.adicionaValor(valor);
        pai.nosFilhos[indice] = no;
    } else {
        for (int i = 0; i < no.getNroChaves(); i++) {
            if (valor == no.chave[i]) {
                return; // Valor duplicado
            }
            if (valor < no.chave[i]) {
                adicionaReg(no.nosFilhos[i], valor, no, i);
                return;
            }
        }
        if (no.getNroChaves() == grau - 1) {
            if (valor == no.chave[grau - 2]) {
                return; // Valor duplicado
            }
            if (valor > no.chave[grau - 2]) {
                adicionaReg(no.nosFilhos[grau - 1], valor, no, grau - 1);
                return;
            }
        } else {
            no.adicionaValor(valor);
            return;
        }
    }
}
```


Phyton

Código 12.1

```
class NoNaria(object):
    def __init__(self, grau):
        self.nroChaves = 0
        self.chave=[None]*(grau)
        self.nosFilhos = [None]*(grau)
        self.grau=grau

    def getNroChaves(self):
        return self.nroChaves

class ArvoreNaria(object):
    def __init__(self, grau):
        self.raiz = None
        self.grau=grau
```

Código 12.2

```
def ordenado1(self):
    aux=self.raiz
    self.ordenado(aux)

def ordenado(self, no):
    if no <> None:
        j=0
        while j < no.nroChaves:
            self.ordenado(no.nosFilhos[j])
            print no.chave[j]
            j+=1
        self.ordenado(no.nosFilhos[j])
```

Código 12.3 e Código 12.4

```
def busca_no1(self, valor):
    if self.raiz == None:
        print "Nao Existe"
        return None # Nao existe o valor
    else:
        return self.busca_no(self.raiz, valor)

def busca_no(self, no, valor):
    if no==None:
        print "Nao Existe"
        return None
    else:
        i=0
        while i < no.getNroChaves():
            if valor == no.chave[i]:
                print "Existe"
                return no
            if valor< no.chave[i]:
                return self.busca_no(no.nosFilhos[i], valor)
            i+=1
        if no.getNroChaves() == self.grau - 1:
            if valor > no.chave[self.grau - 2]:
                return self.busca_no(no.nosFilhos[self.grau - 1], valor)
            else:
                print "Nao Existe"
                return None
        else:
            print "Nao Existe"
            return None
```

Código 12.5

```

def inserir_n_naria(self, valor):
    if self.raiz == None:
        self.raiz = NoNaria(self.grau)
        self.raiz.chave[self.raiz.nroChaves]=valor
        self.raiz.nroChaves+=1
        i=0
        while i < self.grau:
            self.raiz.nosFilhos[i]= NoNaria (self.grau)
            i+=1
    else:
        self.adicionaRec(self.raiz, valor, self.raiz, 0)

def adicionaRec(self, no, valor, pai, indice):
    if no==None:
        no = NoNaria(self.grau)
        no.chave[no.nroChaves]=valor
        no.nroChaves+=1
        i=0
        while i < self.grau:
            if no.nosFilhos[i] == None:
                no.nosFilhos[i]= NoNaria (self.grau)
                i+=1
            pai.nosFilhos[indice] = no
    else:
        i=0
        while i < no.getNroChaves():
            if valor == no.chave[i]:
                return #Valor duplicado
            if valor < no.chave[i]:
                self.adicionaRec(no.nosFilhos[i], valor, no, i)
                return
            i+=1
        if no.getNroChaves() == self.grau - 1:
            if valor == no.chave[self.grau - 2]:
                return
            if valor > no.chave[self.grau - 2]:
                self.adicionaRec(no.nosFilhos[self.grau-1], valor, no, self.grau-1)
                return
        else:
            no.chave[no.nroChaves]=valor
            no.nroChaves+=1
            i=0
            while i < self.grau:
                if no.nosFilhos[i] == None:
                    no.nosFilhos[i]= NoNaria (self.grau)
                    i+=1
            return

```



Para fixar!

1. Utilizando o algoritmo para inserção definido neste capítulo, desenhe numa folha de papel duas árvores N -árias contendo 100 chaves geradas aleatoriamente. Na primeira delas, faça a inserção na ordem gerada aleatoriamente; na segunda, use os mesmos valores gerados aleatoriamente, mas numa ordem de inserção que distribua de modo homogêneo as chaves pelas subárvores, procurando ter uma árvore com a menor altura possível. Compare as duas árvores e procure tirar suas conclusões sobre a importância de ter as chaves distribuídas homogeneamente pelas subárvores.

2. Utilizando as implementações que você fez para os algoritmos apresentados para a remoção de chaves numa árvore N -ária, tanto por meio da colocação de marcas quanto na remoção com rearranjo dos nós da árvore, e usando a função para inserção que estudamos, procure criar a árvore da [Figura 12.7](#) e realizar as remoções ali sugeridas. Teste também as funções de percurso ordenado para verificar a lista de chaves antes e após as remoções.



Para saber mais

Dentro do propósito geral deste livro, que é apresentar de forma clara e numa linguagem acessível uma visão geral básica das principais estruturas de dados existentes, orientamos aqueles que desejam se aprofundar nos estudos sobre árvores N -árias que leiam da Parte V, Capítulos 18, do livro *Algoritmos: teoria e prática* ([Wirth, 1989](#)), e do Item 7.3 do livro *Estruturas de dados usando C* ([Tenenbaum, 2004](#)).



Navegar é preciso

Diferentemente das árvores binárias, quase não existem ferramentas disponíveis na internet para árvores N -árias gerais. O que se encontra, geralmente, são ferramentas para tipos específicos de árvores N -árias, principalmente para árvores B, que serão vistas no Capítulo 13. Assim, deixaremos para relacionar algumas das ferramentas que constam do “Navegar é preciso” do capítulo seguinte. De outro lado, como em computação tudo é dinâmico e novidades surgem a cada minuto, não deixe de procurar por essas ferramentas – quem sabe você encontra algo interessante?

Exercícios

1. Crie uma função de percurso que devolva a lista das chaves que fazem parte de determinado nível da árvore.
2. Faça as alterações necessárias nas funções *busca_no_ins* e *inserir_n_naria* para permitir a inserção de chaves duplicadas.

3. Para o conjunto de dados da [Figura 12.2](#) (02-07-14-18-23-32-45-69-78-82-95-98-99-100-107-116-180-184-190-201-220-231-244-250-301-310-512), usando as funções apresentadas nos Capítulos 11 e 12, construa uma árvore binária e uma árvore N -ária (grau 5), em que em ambas a ordem de inserção das chaves seja gerada de modo aleatório. A partir disso, ajuste a função *busca_no* vista para que conte quantos nós foram visitados na localização de cada chave. Ao final, devolva no programa principal os números médios de nós visitados em ambas as árvores, obtidos na localização de todas as chaves. Tente refazer o exercício, porém, definindo uma ordem de entrada em que cada árvore tenha a menor altura possível. Anote suas conclusões em seu caderno.
4. Crie uma função que devolva a relação de chaves presentes nos nós folhas de uma árvore N -ária.
Conseguiu? Parabéns, você já domina os conceitos fundamentais da estrutura de dados árvore N -ária!!!

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Ao longo deste capítulo, cujo objetivo era mostrar as principais diferenças que as árvores N -árias apresentam em relação ao que foi visto no Capítulo 11, procuramos, de forma sutil, estimular você a perceber que a simples construção de árvores, sem preocupação com a distribuição homogênea das chaves pelas subárvores, pode trazer algum prejuízo em termos de eficiência nas futuras buscas por chaves presentes nessas árvores. Nesse contexto, existe o conceito de *balanceamento* em árvores

de busca, em que, a cada inserção ou remoção de uma chave, procura-se reorganizar a árvore de forma que as chaves continuem distribuídas de forma homogênea pelas subárvores com o intuito de que a árvore, seja binária, seja N -ária, tenha a menor altura possível, para reduzir o número de nós visitados numa eventual busca por determinada chave. É isso que veremos no Capítulo 13.

Árvores balanceadas

“ Se o vento soprar de uma única direção, a árvore crescerá inclinada. ”

PROVÉRBIO CHINÊS

Quando armazenamos dados em uma estrutura de dados árvore, se não houver preocupação com a distribuição homogênea desses dados pelos galhos (subárvores), poderemos ter árvores que crescem apenas para um lado, o que traz prejuízos quando se imagina utilizá-las como estruturas que ofereçam maior eficiência na busca e na recuperação de dados.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender o conceito de balanceamento de dados numa estrutura de dados árvore;
- compreender como manter uma árvore binária balanceada (árvores AVL);
- compreender como manter uma árvore *N*-ária balanceada (árvores B).



Para começar

No Capítulo 12, demos um exemplo no qual havia duas jabuticabeiras. Numa delas, cada galho, a partir da raiz, se dividia em no máximo dois novos galhos; na outra, cada galho podia ser dividido em até cinco novos galhos.

Agora vamos considerar que em ambas teríamos a possibilidade de ter cada galho dividido em até cinco novos galhos. Contudo, na primeira jabuticabeira, no ponto em que ele se divide em novos galhos, temos entre um e quatro frutos, tanto na ponta de cada galho, como no ponto de divisão dos novos galhos; na segunda, temos a mesma condição, porém, com as restrições de que, para haver um novo galho, é necessário que, no ponto onde ele surge, já existam quatro jabuticabas e que sua distribuição seja feita de forma homogênea pelos galhos, ou seja, que as alturas dos galhos que partem da raiz sejam as mesmas. Em ambas, o número de galhos é sempre igual ao número de frutos no ponto onde surgia o galho somado de um.

Nesse contexto, podemos ter a primeira jabuticabeira apresentando galhos com menos frutos em cada ponto de divisão e, conseqüentemente, uma árvore irregular no sentido de diferentes alturas para seus galhos.

Assim, responda: qual deve ser o formato das duas árvores para que cada uma comporte 200 frutos, considerando que na primeira jabuticabeira a distribuição dos frutos se dê sem uniformidade pela árvore (10% na primeira e na segunda subárvores que partem da raiz; 25% na terceira; e 50% na quarta) e na segunda, conforme descrito, a distribuição seja homogênea?

No caso de os frutos serem colhidos, em qual das jabuticabeiras você teria maior dificuldade?



Você deve ter concluído que, na primeira jabuticabeira, teríamos alturas variáveis para os galhos e maior dificuldade de acesso às jabuticabas que estivessem nas pontas dos galhos mais altos. Já na segunda jabuticabeira, teríamos galhos de mesma altura, o que implicaria esforço para alcançar as jabuticabas das pontas, distribuídas de maneira igual. Além disso, conseguiríamos uma árvore mais baixa, com consequente acesso, em menor tempo, a todas as jabuticabas.

Essa distribuição homogênea é a principal característica do que chamamos de *balanceamento* numa estrutura de dados árvore, seja ela binária, seja N -ária. A ideia central do balanceamento é de que, a cada novo elemento armazenado ou removido da árvore, seja feita uma reorganização para que a distribuição dos elementos pelas subárvores continue homogênea, ou seja, como visto na segunda jabuticabeira.

Neste capítulo conheceremos a conceituação de árvores binárias e N -árias balanceadas, por meio de suas representantes mais difundidas, que são: árvores AVL e árvores B. Vamos lá!



Conhecendo a teoria para programar

Nos Capítulos 11 e 12, vimos o funcionamento básico da estrutura de dados árvore, bem como as vantagens que essa estrutura traz quando utilizada em tarefas de busca. Foi possível também entender que árvores com altura menor possibilitam, em média, uma redução no número de acessos aos nós necessários para se localizar um elemento ali armazenado.

Assim, agora nosso objetivo passará a ser estudar árvores nas quais é mantida uma distribuição homogênea dos elementos armazenados pelas subárvores. Para isso, vamos conceituar o balanceamento em estrutura de dados árvore e apresentar o funcionamento das árvores binárias balanceadas (aqui representadas pelas árvores AVL) e das árvores N -árias balanceadas, (mais especificamente das árvores B).

Árvores Binárias Balanceadas: Árvores AVL

Todos os conceitos e definições de árvores binárias vistos no Capítulo 11 valem para as árvores binárias balanceadas (que, daqui em diante, serão tratadas apenas como árvores AVL). As árvores AVL* são árvores binárias em que a distribuição dos elementos é feita respeitando determinadas condições que vão garantir o balanceamento dessa árvore. Numa árvore AVL, o balanceamento é definido a partir das alturas das subárvores nela existentes. Nesse tipo de árvore, a diferença entre as alturas das subárvores esquerda e direita de qualquer nó é de no máximo 1, ou seja, se a altura da subárvore esquerda é N , então, a altura da subárvore direita será igual a N , $N-1$ ou $N+1$.

*O termo AVL é proveniente dos nomes de seus criadores, os matemáticos russos Georgy Adelson-Velsky e Evgeniy Landis.

Para ilustrar, vejamos as árvores da [Figura 13.1](#). Tanto a árvore (a) quanto a árvore (b) são árvores AVL, uma vez que as alturas das subárvores esquerda e direita nunca diferem de 1 entre elas. Já as árvores (c) e (d) não são árvores AVL, pois na árvore (c) a altura da subárvore esquerda do nó raiz difere de 2 da altura de sua subárvore direita, e na árvore (d) a altura da subárvore direita do nó raiz difere de 3 da altura de sua subárvore esquerda.

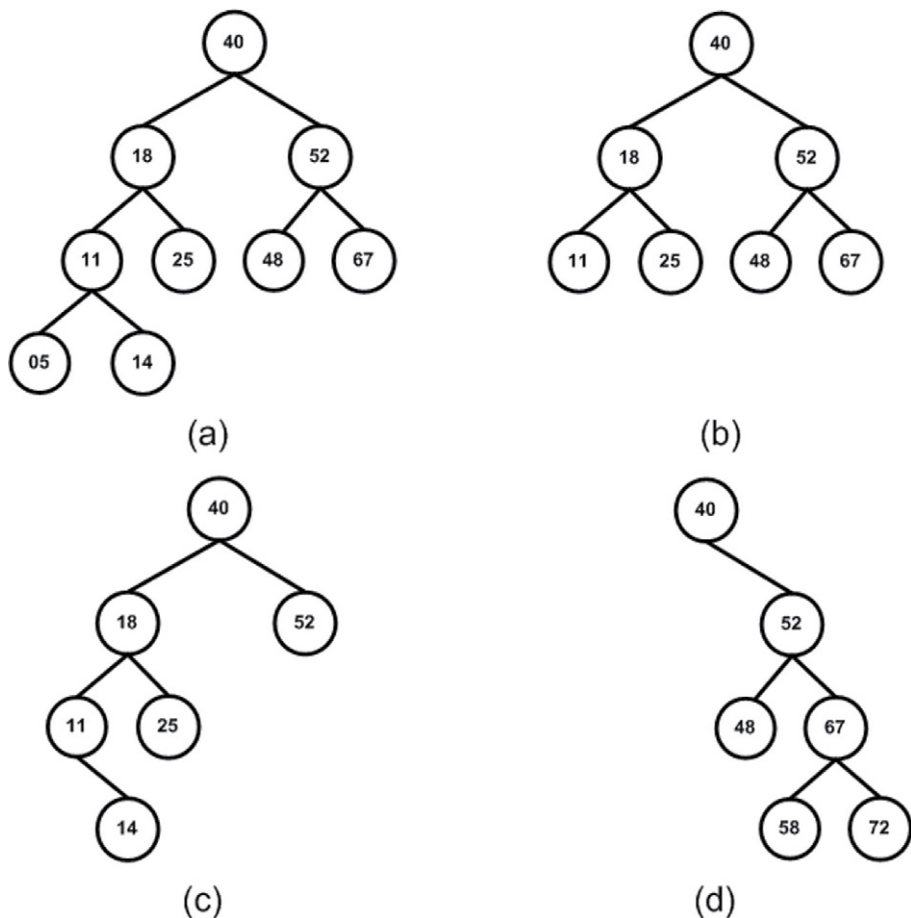


FIGURA 13.1: Árvores AVL e árvores não AVL.

Atenção

O balanceamento por altura não é a única forma de balanceamento existente para a estrutura de dados árvore. Outras formas são encontradas na literatura, entre elas o balanceamento por peso, no qual são observados o número de nós nulos de cada subárvore e o balanceamento de Tarjan, em que para todos os nós da árvore o comprimento do maior segmento entre esse nó e um de seus descendentes folha é, no máximo, o dobro do comprimento do menor segmento entre esse nó e seus descendentes folha.



Embora as operações básicas de percurso (busca), inserção e remoção em árvores AVL sejam as mesmas de uma árvore binária (busca) qualquer, para que seja garantido o balanceamento, torna-se necessário que, após a inserção ou remoção de um elemento, seja verificado se a condição de balanceamento por altura se mantém e se a ordenação das chaves não é afetada.



Dica

Assim como adotado para as árvores binárias, para armazenar os dados secundários dos elementos que farão parte do nó da árvore AVL, bastaria que tivéssemos mais uma lista de N posições. Porém, por motivo de simplificação, ficaremos restritos apenas ao armazenamento das chaves como representantes do elemento armazenado.

No caso das árvores AVL, para encontrar a altura de uma subárvore, temos a implementação da função *altura*, em que usamos as mesmas declarações vistas para uma árvore binária no Capítulo 11, e R é a raiz da subárvore.

Código 13.1

```
int altura(tipo_no *R)
{
    int Alt_esq, Alt_dir;
    if (R == NULL)
        return -1;
    else
    {
        Alt_dir = altura(R->dir);
        Alt_esq = altura(R->esq);
        if (Alt_dir > Alt_esq)
            return Alt_dir+1;
        else
            return Alt_esq+1;
    }
}
```

Ainda no contexto do que foi visto para a implementação de uma árvore binária no Capítulo 11, temos as funções *inserir_binaria* e *remover_binaria*. Usando essas funções e retomando a árvore da [Figura 13.1\(a\)](#), procure assinalar quais inserções, nessa árvore, manteriam o balanceamento.

Terminou? Então, confira na [Figura 13.2](#) se você acertou!

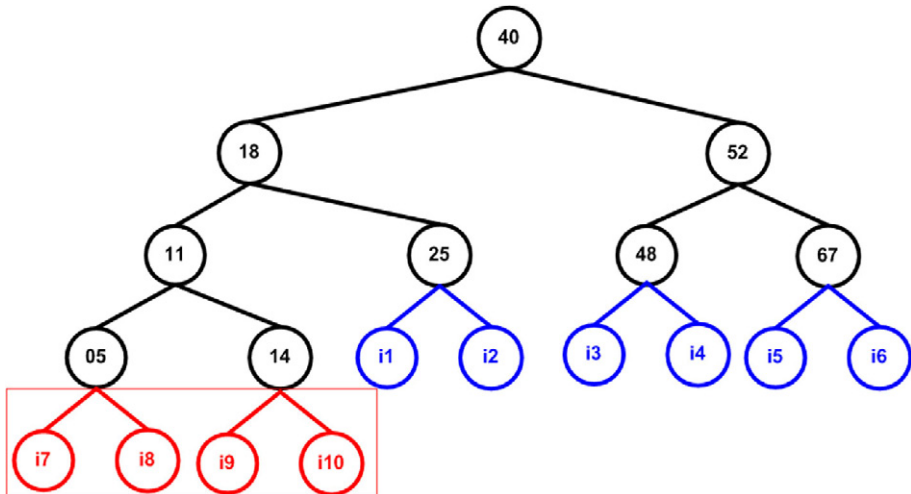


FIGURA 13.2: Inserções possíveis na árvore.

É possível verificar que as inserções individuais i1, i2, i3, i4, i5 ou i6 levariam a situações em que o balanceamento seria mantido. De outro lado, as inserções individuais i7, i8, i9 ou i10 provocariam um desbalanceamento na árvore.

Assim como foi feito para a inserção, verifique, individualmente, quais remoções dos elementos da árvore da [Figura 13.1\(a\)](#) manteriam a árvore balanceada e quais levariam ao desbalanceamento, conferindo os resultados a seguir:

Remoções que não promoveriam o desbalanceamento: 05-11-14-18-48-52-67

Remoções que promoveriam o desbalanceamento: 25-40

Pelo exemplo, percebemos que nas situações em que se tem o desbalanceamento da árvore a partir de uma inserção ou remoção, algo precisa ser feito para restabelecer a diferença de 1 entre as alturas das subárvores esquerda e direita dos nós da árvore, sem que a ordenação das chaves seja afetada.

É aqui que entra o conceito de rotação de nós numa árvore, que objetiva o restabelecimento do balanceamento da árvore após a inserção ou a remoção de um elemento que tenha provocado um desequilíbrio na distribuição dos elementos, de modo que a condição de balanceamento não seja mais atendida.

Para compreender melhor esse conceito, tomemos o exemplo da árvore da [Figura 13.3\(a\)](#). Note que a inserção da chave 35 na árvore leva a seu desbalanceamento, uma vez que a diferença de altura entre as subárvores do nó raiz é superior a 1. Para restabelecer o balanceamento, poderia ser feita uma rotação dos nós, conforme ilustrado na [Figura 13.3\(b\)](#).

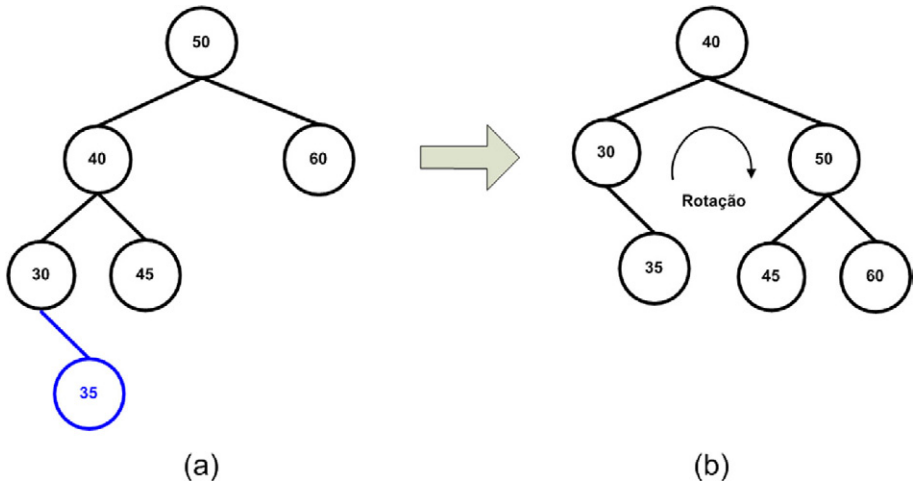


FIGURA 13.3: Rotação à direita numa árvore AVL.

De modo geral, podemos resumir as rotações de nós em árvores AVL em:

- **rotação à direita:** neste tipo de rotação, o nó raiz da subárvore é deslocado para a posição de seu nó filho à direita, que, por sua vez, continua a ser o nó filho à direita do nó deslocado. O nó filho à direita do nó filho à esquerda da raiz é deslocado para ser o nó filho à esquerda do nó raiz deslocado. O nó filho à esquerda do nó raiz ocupa seu antigo lugar, passando a ser a nova raiz.

Considerando as declarações feitas para a árvore binária, no Capítulo 11, veja como ficaria a implementação dessa rotação:

Código 13.2

```
void rotacao_direita(no **R)
{
    no *Aux;
    if (*R != NULL)
    {
        Aux = (*R)->esq;                // Aux aponta o nó filho à esquerda da raiz da subárvore
        (*R)->esq = Aux->dir;            // Nó filho à direita de Aux passa a ser o nó filho à esquerda da raiz
        Aux->dir = *R;                   // Raiz passa a ser o nó filho à direita de Aux
        *R = Aux;                       // Aux passa a ser a nova Raiz da subárvore
    }
}
```

O exemplo da [Figura 13.3](#) ilustra uma rotação à direita.

- **rotação à esquerda:** neste tipo de rotação, o nó raiz da subárvore é deslocado para a posição de seu nó filho à esquerda, que, por sua vez, continua a ser o nó filho à esquerda do nó deslocado. O nó filho à esquerda do nó filho à direita da raiz é deslocado para ser o nó filho à direita do nó raiz deslocado. O nó filho à direita do nó raiz ocupa o seu antigo lugar, passando a ser a nova raiz.

Veja como ficaria a implementação dessa rotação:

Código 13.3

```
void rotacao_esquerda(no **R)
{
    no *Aux;
    if (*R != NULL)
    {
        Aux = (*R)->dir;           // Aux aponta o nó filho à direita da raiz da subárvore
        (*R)->dir = Aux->esq;       // Nó filho à esquerda de Aux passa a ser o nó filho à direita da raiz
        Aux->esq = *R;              // Raiz passa a ser o nó filho à esquerda de Aux
        *R = Aux;                  // Aux passa a ser a nova Raiz da subárvore
    }
}
```



Conceito

A aplicação das rotações, além de objetivar o restabelecimento do balanceamento numa árvore AVL, deve preservar a ordenação prévia das chaves (elementos da árvore).

Para ilustrar a *rotação à esquerda*, vejamos a [Figura 13.4](#). Aplicando na árvore (a) as ações definidas para essa rotação, temos a árvore (b), que atende tanto a manutenção da ordenação entre as chaves quanto a condição de balanceamento por altura.

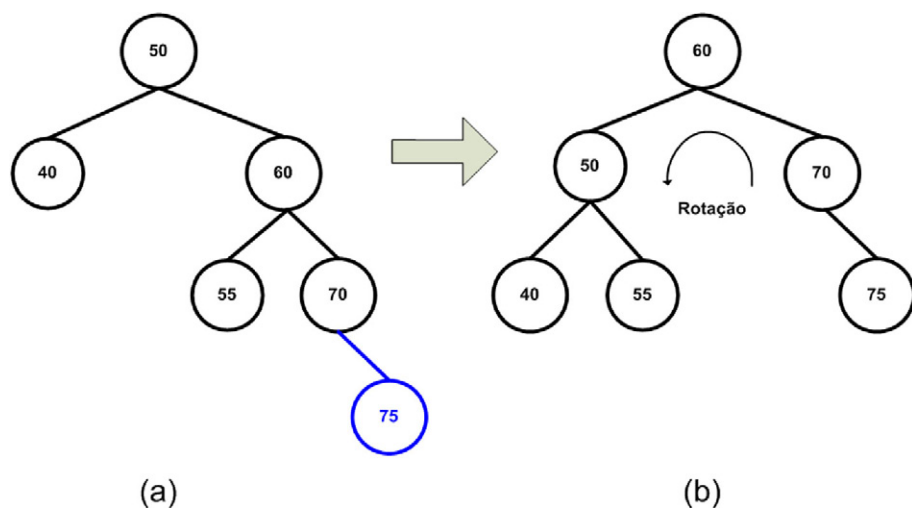


FIGURA 13.4: Rotação à esquerda numa árvore AVL.

Atenção

Existem situações nas quais necessitamos de duas rotações seguidas (uma à direita e outra à esquerda, ou uma à esquerda e outra à direita) para que seja restabelecido o balanceamento da árvore. Essas rotações em sequência são conhecidas como “rotações duplas”.



Na [Figura 13.5](#) temos uma situação em que é necessária a aplicação de uma “rotação dupla” para restabelecer o balanceamento da árvore. Ao tentar inserir a chave 50 na árvore da [Figura 13.5\(a\)](#), teremos que a diferença de altura entre as subárvores direita e esquerda da raiz será 2 (árvore desbalanceada). Nesse caso, uma simples rotação não seria suficiente para restabelecer o balanceamento. Assim, é necessário primeiro aplicar uma rotação à esquerda, a partir do nó com chave 40, obtendo a árvore da [Figura 13.5\(b\)](#) e, na sequência, uma rotação à direita a partir da raiz, obtendo a árvore da [Figura 13.5\(c\)](#), que atenderá novamente às propriedades de uma árvore AVL.

De forma resumida, podemos dizer que as operações de inserção e de remoção de nós numa árvore AVL são as mesmas usadas em árvores binárias; a diferença é que, após a realização dessas operações, torna-se necessário verificar a manutenção do balanceamento por altura. No caso de se identificar um desbalanceamento na árvore, ou seja, se a diferença for maior

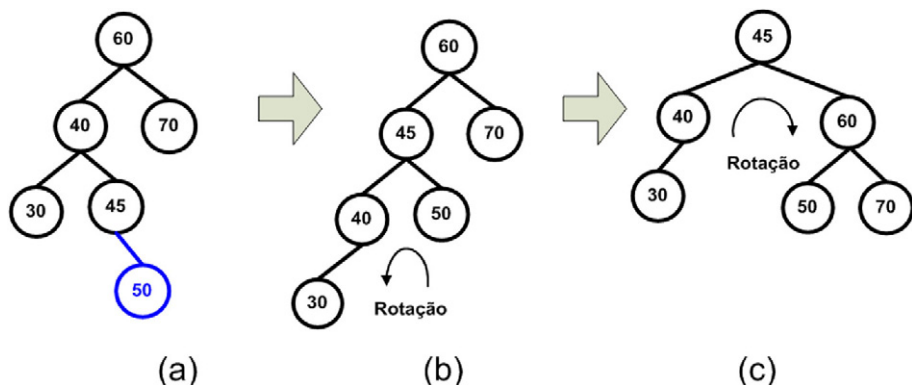


FIGURA 13.5: Balanceamento de uma árvore AVL usando “rotação dupla”.

que 1 (ou menor que -1) entre as alturas das subárvores direita e esquerda, torna-se necessária a aplicação de rotações à direita, à esquerda ou “rotações duplas”, dependendo do caso, para que se restabeleça o balanceamento da árvore. Lembramos que as rotações devem preservar a ordenação das chaves armazenadas na árvore.

Para aprofundamento, tente implementar a função que verifica o balanceamento da árvore após a realização de uma inserção ou de uma remoção.



Dica

Para verificar o balanceamento de uma árvore AVL, procure alterar o registro do nó para que ele contenha também um campo no qual possa ser armazenado o fator de balanceamento para determinado nó. Esse fator de balanceamento seria o resultado da diferença entre as alturas das subárvores direita e esquerda do nó.

No caso de dificuldade para a finalização da tarefa, lembre-se de que implementações completas de árvores AVL são fáceis de se encontrar na internet.

Árvores *N*-árias Balanceadas: Árvores B

Assim como ocorre com as árvores AVL em relação às árvores binárias, as árvores B (que são árvores *N*-ária balanceadas) estão sujeitas às mesmas definições e conceitos vistos para as árvores *N*-árias. Mais uma vez, a diferença reside no fato de que a distribuição dos elementos pela árvore é feita respeitando-se determinadas condições que vão garantir seu balanceamento.

Definição

Como visto em [Cormen \(2012\)](#), para que uma árvore qualquer seja considerada uma árvore B, existem propriedades que devem ser respeitadas:

- todo nó R de uma árvore B deve conter um campo (NC), que armazena o número de chaves (N) contidas no nó; uma lista (LC), que armazena as N possíveis chaves, que devem estar ordenadas de forma crescente ($Ch_1 \leq Ch_2 \leq Ch_3 \leq \dots \leq Ch_N$); uma lista (AP) com apontadores para seus $N+1$ possíveis nós filhos; e um campo (F); que indica se o nó é ou não um nó folha;
- os nós internos (aqueles que não são folhas) devem possuir $N+1$ apontadores para seus possíveis nós filhos. Nós folhas não têm filhos;
- as chaves armazenadas em um nó R separam as faixas de chaves armazenadas nas subárvores desse nó, em que na i -ésima subárvore do nó R sempre há chaves com valores menores que a i -ésima chave do nó R , e na $(i+1)$ -ésima subárvore sempre há chaves maiores que a i -ésima chave do nó R ;
- todos os nós folhas estão no mesmo nível da árvore;
- existem limites inferiores e superiores para o número de chaves em cada nó. Esses limites são caracterizados pelo grau mínimo da árvore G , que deve ser ≥ 2 . Todo nó, exceto a raiz, tem no mínimo $G-1$ chaves; consequentemente, todo nó interno, exceto a raiz, tem no mínimo G nós filhos. Sendo a árvore não vazia, sua raiz terá no mínimo uma chave. De outro lado, todo nó poderá ter até $2G-1$ chaves, tendo os nós internos no máximo $2G$ nós filhos.

Para exemplificar, vejamos o exemplo da [Figura 13.6](#), em que temos uma árvore B de grau 2. Podemos notar que todas as propriedades são encontradas nessa árvore.

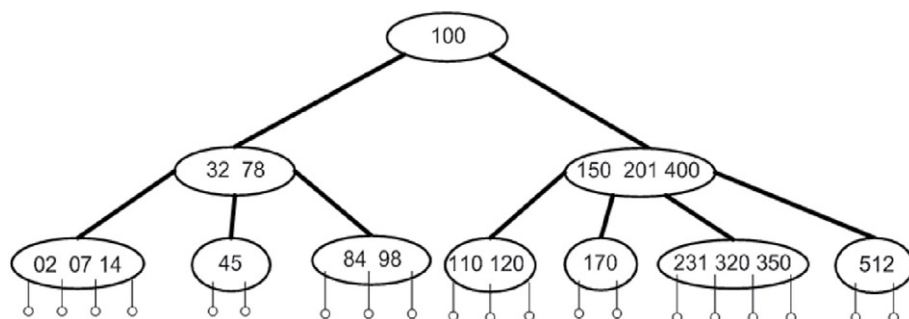


FIGURA 13.6: Exemplo de uma árvore B.

Conforme descrito anteriormente, a árvore B, por ser um tipo de árvore *N*-ária balanceada, favorece o armazenamento de maior volume de dados, sem um impacto significativo no número de acessos aos nós quando se deseja localizar algum elemento (chave). Isso decorre do fato de o armazenamento de vários elementos ser feito num mesmo nó, e também por ser balanceado de modo que todos os nós folha estejam num mesmo nível.

Assim, o conceito de *altura* é importante para essa árvore, pois, quanto menor for sua altura, menor será o número de acessos aos nós necessários para localizar um de seus elementos.

Considerando-se a situação em que o número de nós $N \geq 1$, para qualquer árvore B de grau mínimo $G \geq 2$, sua altura H é representada da seguinte forma:

$$H \leq \log_g (n + 1)/2$$

Essa altura reduzida, se comparada às outras árvores estudadas, mostra o potencial desse tipo de árvore em situações que envolvam a busca em grande volume de dados e, principalmente, quando ela está armazenada em dispositivos de maior lentidão, como o disco e outros dispositivos de armazenamento secundário.



Dica

As árvores B e suas variações (árvores B+ e árvores B*) são muito utilizadas em sistemas de banco de dados.

Implementação de uma árvore B

O primeiro passo para implementar uma árvore B consiste na definição do grau mínimo (G) da árvore, que é o número mínimo de nós filhos que os nós (exceto a raiz) de uma árvore B terão. A partir daí, vamos definir o registro que representará o nó desse tipo de árvore.

Assim como descrito para uma árvore *N*-ária, a forma comumente adotada para a implementação de uma árvore B se dá pelo uso de alocação dinâmica de memória.

No contexto de alocação dinâmica, é possível especificar que a estrutura do registro nó vai conter: um valor numérico, em que é armazenado o número de chaves existentes no nó (o máximo será $2 \cdot G - 1$); um campo que indicará se é um nó folha; uma lista de $2 \cdot G - 1$ posições, que armazenará as chaves; e uma lista de $2 \cdot G$ posições, que armazenará os apontadores para as subárvores.

Assim, temos na [Figura 13.7](#) a representação de um nó dessa árvore:

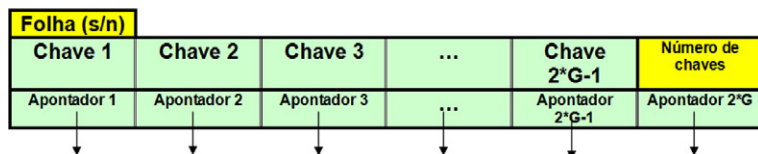


FIGURA 13.7: Representação de um nó da árvore B.

Declaração de uma árvore B

Consiste na definição do nó representado na [Figura 13.7](#) por meio de um registro. Para representar os campos descritos na representação desse nó, temos: *nro_chaves* (armazena o número de chaves que estão armazenadas no nó em determinado momento); *folha* (indica se o nó está como uma folha da árvore); *chaves* (vetor com 2^G-1 posições que armazena as chaves); e *apontadores* (vetor com 2^G posições que armazena os apontadores para as subárvores).

Código 13.4

```
#define G 2 // O valor 2 foi escolhido como exemplo para o Grau Mínimo
typedef struct tipo_no no;
struct tipo_no
{
    int nro_chaves;
    int folha;
    tipo_dado chaves[2*G-1];
    tipo_no *apontadores[2*G];
};
```

Na declaração acima, foi usado um exemplo com o valor de grau mínimo igual a 2. Assim, todo nó diferente da raiz terá dois, três ou quatro filhos. Essa árvore é muito utilizada para o aprendizado de árvore B e é conhecida como árvore 2-3-4.

Como nas árvores *N*-árias, nos exemplos que serão apresentados sobre árvores B vamos também adotar que *tipo_dado* será um valor do tipo *inteiro*. Assim:

```
typedef int tipo_dado;
```

**Dica**

Assim como adotado para as árvores N -árias, para armazenar os dados secundários dos elementos que farão parte do nó da árvore B, bastaria que tivéssemos mais uma lista de N posições. Porém, por motivo de simplificação, mais uma vez ficaremos restritos ao armazenamento das chaves como representantes do elemento armazenado.

Para criar a raiz inicial de uma árvore B, bastaria alocar um nó e inicializá-lo com os valores necessários para cada campo (número de chaves igual a zero), indicá-lo como um nó folha e atribuir a esse nó o apontador de raiz da árvore. A seguir, temos a implementação da função *criar_arvore_B*, que realiza essa tarefa.

Código 13.5

```
void criar_arvore_B(no **A)
{
    no *Novo;

    Novo = (no *)malloc(sizeof(no)); // Aloca espaço na memória correspondente ao nó Novo
    Novo->folha = 1;                  // Coloca status de folha como sim
    Novo->nro_chaves = 0;              // Acerta o número de chaves de Novo
    *A = Novo;                        // Novo passa a ser a raiz da árvore
}
```

Operações básicas numa árvore B

A manipulação de uma árvore B é possível por meio de diferentes operações. Assim como fizemos com as árvores N -árias, vamos focar nas operações básicas de maior utilidade e difusão: percurso (busca), inserção e remoção.

Percurso de árvore B

Conforme já descrito, a árvore B é uma árvore N -ária com a restrição de ser balanceada. Dessa forma, as operações de percurso e de busca por determinado elemento (chave) são idênticas às apresentadas no Capítulo 12 para as árvores N -árias, e não serão repetidas aqui.

Inserção de um elemento (chave) numa árvore B

Quando abordamos a inserção numa árvore B, adotamos uma ideia similar à usada nas árvores N -árias, vistas no Capítulo 12, no que se refere à

distribuição das chaves de forma ordenada, por meio da lista de chaves de cada nó e de seus nós filhos.

Porém, conforme visto em [Cormen \(2012\)](#), a inserção deve oferecer condições para que as propriedades descritas para uma árvore B sejam mantidas, e essa não é uma tarefa das mais simples. Um caminho possível seria procurar um nó folha R em que a chave se encaixasse, segundo a ordenação existente. Caso o nó folha R encontrado para inserção esteja cheio (número máximo $2 \cdot G - 1$ de chaves), torna-se necessária uma reorganização que libere o espaço necessário para a nova chave. Essa reorganização consiste em repartir o nó cheio R em torno de sua chave mediana M , obtendo dois nós, R_1 e R_2 , cada qual contendo $G - 1$ chaves. A chave mediana é deslocada para o nó pai Pr de R , assim como a chave que ficará entre os nós filhos R_1 e R_2 . Mas pode ocorrer de o nó pai Pr também já estar cheio. Nesse caso, devemos repetir para Pr o mesmo processo feito para o nó R . Essa situação pode levar a divisões sucessivas da árvore anterior, até que se encontre um nó pai com espaço para inserção da chave deslocada.

Outra forma utilizada para permitir que a inserção ocorra numa única passagem pela árvore, no sentido raiz-folha, é aquela em que, ao descer pela árvore em busca da posição de inserção, já seja feita a repartição dos nós cheios que estiverem no caminho percorrido. Dessa forma, ao precisar repartir o nó folha cheio R , já saberíamos que seu pai não está cheio.

Para compreender melhor o processo que reparte um nó que estava com número máximo de chaves, veja a [Figura 13.8](#), em que o nó X será dividido em dois novos nós e a chave mediana 40 subirá para o nó pai de X .

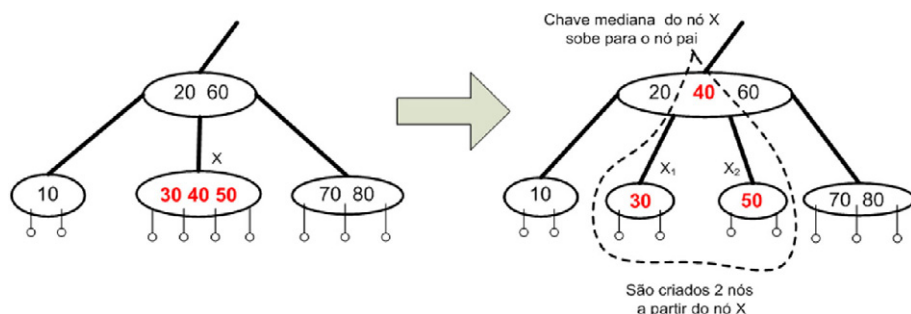


FIGURA 13.8: Repartição de um nó numa árvore B.

Note que repartir o nó cheio X significa dividi-lo nos nós X_1 e X_2 , subindo a chave mediana para o nó pai de X .

Esse processo está implementado na função *repartir*, apresentada a seguir. Note que o primeiro elemento do vetor está na posição zero (0), daí a necessidade de correções nas faixas e posições utilizadas nas funções apresentadas para árvore B.

Código 13.6

```
void repartir(no **Pai, int i)
{
    no *Ant, *Novo;           // Apontadores para os nós que receberão G-1 chaves (cada um) do nó repartido.
                              // Ant e Novo serão o resultado da repartição de Ant (que é o k-ésimo filho de Pai)

    int k;

    Ant = (*Pai)->apontadores[i]; // Ant aponta o nó filho de Pai que será repartido
    Novo = (no *)malloc(sizeof(no)); // Aloca espaço na memória correspondente ao nó Novo
    Novo->folha = Ant->folha; // Coloca os 2 nós (resultados da divisão com o mesmo status de folha (0 ou 1))
    for (k = 0; k <= G-2; k++) // Copia as G-1 maiores chaves de Ant em Novo
        Novo->chaves[k] = Ant->chaves[k+G];
    if (Ant->folha == 0)
        for (k = 0; k <= G-1; k++) // Copia as G subárvores mais à direita de Ant em Novo
            Novo->apontadores[k] = Ant->apontadores[k+G];
    Novo->nro_chaves = Ant->nro_chaves = G-1; // Acerta o número de chaves de Ant e Novo
    for (k = (*Pai)->nro_chaves-1; k >= i; k--) // Abre espaço para subir a Chave mediana para o nó Pai
        (*Pai)->chaves[k+1] = (*Pai)->chaves[k];
    (*Pai)->chaves[i] = Ant->chaves[G-1]; // Sobe a chave mediana
    ((*Pai)->nro_chaves)++; // Incrementa o número de chaves de Pai
    for (k = ((*Pai)->nro_chaves); k >= i+1; k--) // Abre espaço para a subárvore formada pelo nó Novo
        (*Pai)->apontadores[k+1] = (*Pai)->apontadores[k];
    (*Pai)->apontadores[k+1] = Novo;
}
```

Na função *repartir* temos que Pai é o nó pai de Ant, que será o nó repartido. No nó Ant permanecerão as G-1 menores chaves, e as G-1 maiores serão colocadas no novo nó criado (Novo). Da mesma forma, temos que no nó Ant permanecerão as G subárvores mais à esquerda, e as G subárvores à direita serão deslocadas para o nó Novo.



Atenção

É importante notar que a única forma de aumentar a altura de uma árvore B é repartindo seu nó raiz, no caso de esse nó estar cheio. Assim, uma árvore B cresce em altura a partir de sua raiz, e não de suas folhas.

Retomando a inserção em uma única passagem, temos que considerar também a situação em que a própria raiz já está cheia. Nessa situação existe a necessidade de aumentar a altura da árvore. Para isso, será criado um novo nó raiz que receberá a chave mediana da antiga raiz; esta, por sua vez, será repartida em dois nós, que serão os filhos da nova raiz recém-criada.

A partir dessas considerações, ainda no contexto de inserção em uma única passagem encontrado em [Cormen \(2012\)](#), temos a função *inserir_B*, que permite a inserção de uma chave numa árvore B (e esta se mantém balanceada).

Código 13.7

```
void inserir_B(no **A, tipo_dado Ch)
{
    no *Aux, *Novo;

    Aux = *A;                                     // Aponta a raiz da árvore A
    if (Aux->nro_chaves == 2*G-1)                 // Nó raiz está cheio
    {
        Novo = (no *)malloc(sizeof(no));         // Aloca espaço na memória correspondente ao nó Novo
        Novo->folha = 0;                          // Coloca status de folha como não
        Novo->nro_chaves = 0;                     // Acerta o número de chaves de Novo
        Novo->apontadores[0] = Aux;               // Coloca a antiga raiz como filha de Novo
        *A = Novo;                               // Novo passa a ser a nova raiz
        repartir(&Novo, 0);                       // Repartição da antiga raiz (Aux)
        inserir_chave_B(&Aux, Ch);                // Insere Ch a partir do nó que foi repartido
    }
    else inserir_chave_B(&(*A), Ch);              // Insere Ch a partir da raiz
}
```

Na função *inserir_B*, inicialmente é verificada se a raiz *A* está cheia; se estiver, é criado um novo nó (Novo), que será inicializado e passará a ser a nova raiz da árvore. O passo seguinte reside na repartição da antiga raiz *Aux* em dois nós (*Aux* e mais um nó criado pela função *repartir*), que serão os nós filhos da nova raiz (Novo) para *A*. Por fim, a chave *Ch* é inserida a partir do nó Novo, que é a nova raiz. Caso a raiz *A* não esteja inicialmente cheia, a chave *Ch* é inserida a partir do nó raiz *A*.

Essa inserção é feita pela função *inserir_chave_B*, apresentada a seguir:

Código 13.8

```
void inserir_chave_B(no **N, tipo_dado Ch)
{
    no *Aux;                      // Apontador usado como k-ésimo filho de N
    int k;

    k = (*N)->nro_chaves-1;       // Indica a última posição ocupada em chaves[k] no nó N
    if ((*N)->folha == 1)         // Verifica se N é nó folha (para receber Ch)
    {
        while (k >= 0 || Ch < (*N)->chaves[k])
        {
            (*N)->chaves[k+1] = (*N)->chaves[k]; // Abre espaço em N para inserção de Ch
            k--;
        }
        (*N)->chaves[k+1] = Ch; // Insere Ch em N
        ((*N)->nro_chaves)++; // Incrementa o número de chaves de N
    }
    else // Como N não é folha, procura filho de N para inserção
    {
        while (k >= 0 || Ch < (*N)->chaves[k])
            k--; // Procura subárvore de N para inserção de Ch
        k++;
        Aux = (*N)->apontadores[k]; // Aux aponta subárvore para inserção de Ch
        if (Aux->nro_chaves == 2*G-1) // Verifica se Aux (filho de N) está cheio
        {
            repartir(&(*N), k); // Repartição de Aux
            if (Ch > (*N)->chaves[k]) // Identifica em qual dos nós gerados da repartição de Aux deverá inserir Ch
                k++;
            Aux = (*N)->apontadores[k];
        }
        inserir_chave_B(&Aux, Ch); // Insere Ch na subárvore com raiz Aux
    }
}
```


Na função *inserir_chave_B*, inicialmente é verificado se o nó *N* é um nó folha. Caso *N* seja uma folha, então insere Ch de forma ordenada nesse nó. No caso de não ser um nó folha, então vai procurar uma subárvore de *N* para inserção. Nessa subárvore, verifica se sua raiz Aux está cheia. Em caso afirmativo, reparte Aux em dois novos nós e identifica em qual subárvore iniciada por um desses nós deverá tentar inserir Ch, deixando essa subárvore apontada por Aux. Procura inserir Ch na subárvore de raiz Aux.

Para ilustrar o funcionamento da inserção, tomemos o exemplo da [Figura 13.9](#). Nele, desejamos inserir a chave 320 na árvore B ($G = 2$) ali representada.

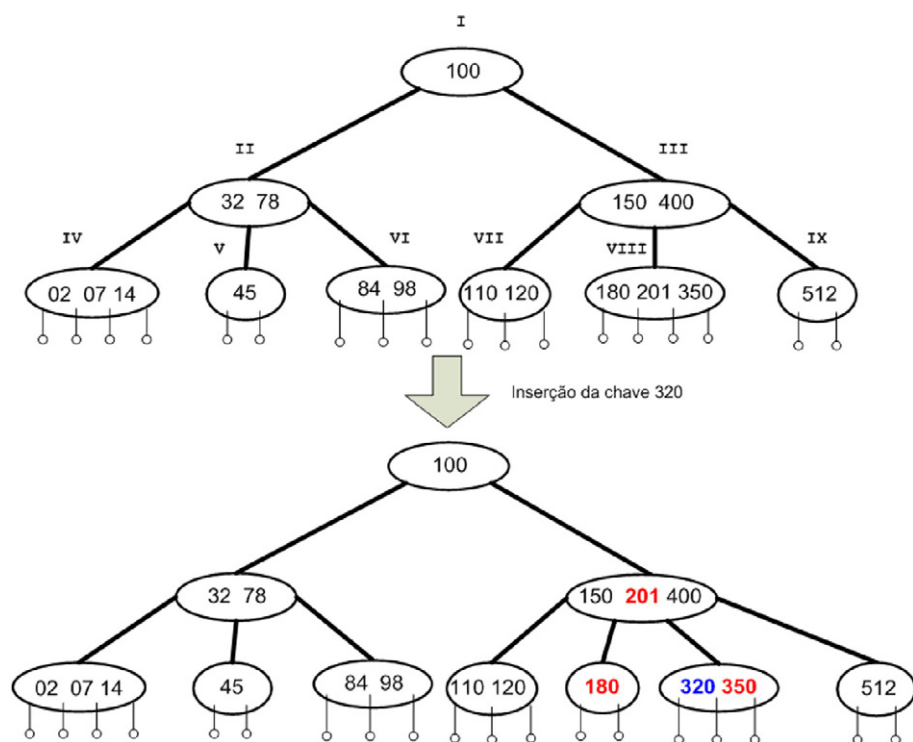


FIGURA 13.9: Inserção de uma chave numa árvore B.

Visita: nó I, chave 100

Não insere 320, porque o nó I não é nó folha

Visita: nó III, chaves 150-400

Não insere 320, porque o nó III não é nó folha

Visita: nó VIII, chaves 180-201-350

Não existe espaço no nó VIII

Nó VIII é repartido e sua chave mediana (201) sobe para o nó pai

Dois nós substituem o nó VIII

A chave 320 é inserida no segundo nó originado pela repartição do nó VIII

Remoção de um elemento (chave) de uma árvore B

Ao pensarmos na remoção de elementos de uma árvore B, precisamos destacar que, diferentemente da inserção, em que as chaves são inseridas apenas em nós folhas, na remoção existe a possibilidade de remover chaves de qualquer um dos nós da árvore. Quando a remoção é de uma chave de um nó folha, basta remover a chave e ajustar o número de chaves do nó. De outro lado, quando a remoção é de uma chave de um nó interno, torna-se necessária a verificação da manutenção das propriedades inerentes à árvore B, o que nem sempre é uma tarefa simples.

Assim, após a remoção de uma chave, deve-se garantir que os nós não fiquem com um número pequeno de chaves. O limite estabelecido está relacionado ao grau mínimo (G), em que o número mínimo de chaves em um nó, que não seja a raiz, deverá ser de $G-1$ chaves. Para garantir as propriedades, existem diferentes maneiras de realizar a remoção. Adotaremos aqui a solução encontrada em [Cormen \(2012\)](#), que propõe o seguinte algoritmo (adaptado a nossa nomenclatura) para a remoção de chaves de uma árvore B:

1. se a chave Ch está no nó N , e N é um nó folha, elimine a chave Ch de N ;
2. se a chave Ch está no nó N , e N é um nó interno, faça:
 - a. se o filho Fa que precede a chave Ch no nó N tem no mínimo G chaves, então encontre a chave predecessora Ch' de Ch na subárvore com raiz em Fa . Elimine recursivamente a chave Ch' e

- substitua a chave Ch por Ch' em N (podemos encontrar a chave Ch' e eliminá-la em uma única passagem descendente);
- b. se Fa tiver menos que G chaves, então, simetricamente, examine o filho Fb que segue Ch no nó N . Se Fb tiver no mínimo G chaves, encontre a chave sucessora Ch' de Ch na subárvore com raiz em Fb . Elimine a chave Ch' recursivamente e substitua a chave Ch por Ch' em N (podemos encontrar a chave Ch' e eliminá-la em uma única passagem descendente);
 - c. caso contrário, Fa e Fb têm apenas $G - 1$ chaves, junte a chave Ch e todo o Fb com Fa , de modo que N perde a chave Ch e também o ponteiro para Fb , e agora Fa contém $2G - 1$ chaves. Em seguida, libere Fb e elimine recursivamente Ch de Fa .
3. se a chave Ch não estiver presente no nó interno N , determine a subárvore Sn adequada que deve conter a chave Ch . Se a raiz Rsn desta subárvore tiver somente $G - 1$ chaves, execute a etapa 3a ou 3b conforme necessário para garantir que descendermos até um nó que contém no mínimo G chaves. Então, termine executando recursão no filho adequado de N :
 - a. se Sn tiver somente $G - 1$ chaves, mas tiver um irmão imediato com no mínimo G chaves, dê a Rsn uma chave extra passando uma chave de N para baixo até Rsn , passando uma chave do irmão imediato à esquerda ou à direita de Rsn para cima até N , e passando o ponteiro de filho adequado do irmão para Rsn ;
 - b. se Rsn e seus irmãos imediatos têm $G - 1$ chaves, junte Rsn com um irmão, o que envolve passar uma chave de N para baixo até o novo nó resultante da junção, que assim se torna a chave mediana para esse nó.

Para exemplificar o funcionamento desse algoritmo, vejamos o exemplo apresentado na [Figura 13.10](#).

Observando a [Figura 13.10](#), temos que a remoção da chave 14 leva a situação mais comum, que é remover uma chave que está num nó folha, bastando para isso efetivar sua remoção desse nó. Na remoção da chave 78, existe a necessidade de agrupar seus nós filhos. Por fim, na remoção da chave 400 existe a necessidade de subir a maior chave da subárvore à esquerda para seu lugar. Existem outras possibilidades de remoção que promovem novas situações, sendo todas elas contempladas no algoritmo apresentado.

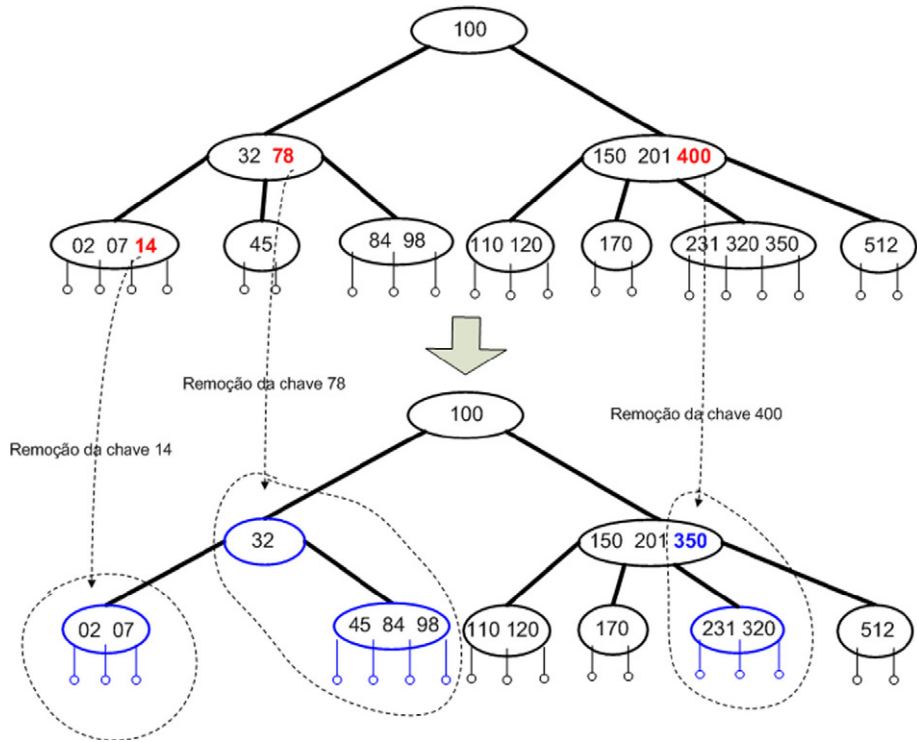


FIGURA 13.10: Remoção de uma chave de uma árvore B.

Atenção

Como a maioria das chaves está nos nós folhas, em geral a remoção torna-se um simples processo de eliminação de uma chave do nó. Porém, quando a remoção ocorre num nó interno, aí o processo exige as ações descritas. Mas, independentemente da aparente complexidade do conjunto de ações necessárias, o tempo de resposta não sofrerá impacto significativo, devido ao número reduzido de nós acessados.

Deixaremos como aprofundamento a possibilidade de implementação do algoritmo descrito para a remoção numa árvore B.



Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos nas seções seguintes implementações nas linguagens de programação Java e Python.

Java

Código 13.1

```
public void alturaArvore() {  
    System.out.println(altura(raiz));  
}  
  
public int altura(No no)  
{  
    if(no == null)  
        return -1;  
  
    int left = altura(this.arvoreEsquerda.raiz);  
    int right = altura(this.arvoreDireita.raiz);  
  
    if(left > right)  
        return left + 1;  
    else  
        return right + 1;  
}
```

Código 13.2

```
public void rotacao_direita(){  
    if(raiz != null) {  
        ArvoreBinaria aux = arvoreEsquerda;  
        arvoreEsquerda = arvoreDireita;  
        arvoreDireita.raiz = raiz;  
        raiz = aux.raiz;  
    }  
}
```

Código 13.3

```
public void rotacao_esquerda(){  
    if(raiz != null) {  
        ArvoreBinaria aux = arvoreDireita;  
        arvoreDireita = arvoreEsquerda;  
        arvoreEsquerda.raiz = raiz;  
        raiz = aux.raiz;  
    }  
}
```

Código 13.4

```
public class NoB {  
  
    private final static int GRAU = 2;  
    private int nro_chaves;  
    public int[] chaves;  
    public NoB[] apontadores;  
    public int folha;  
  
    public NoB() {  
        apontadores = new NoB[2 * GRAU];  
        chaves = new int[2 * GRAU - 1];  
    }  
}
```

Código 13.5

```
public static void criar_arvore_B(NoB no){  
    NoB novo = new NoB();  
    novo.folha = 1;  
    novo.nro_chaves = 0;  
    no = novo;  
}
```

Código 13.6

```
public static void repartir(NoB pai, int i){  
    NoB ant, novo;  
    int k;  
  
    ant = pai.apontadores[i];  
    novo = new NoB();  
    novo.folha = ant.folha;  
    for (k = 0; k <= GRAU-2; k++){  
        novo.chaves[k] = ant.chaves[k+GRAU];  
    }  
    if(ant.folha == 0){  
        for (k = 0; k <= GRAU-1; k++){  
            novo.apontadores[k] = ant.apontadores[k+GRAU];  
        }  
    }  
    novo.nro_chaves = ant.nro_chaves = GRAU-1;  
    for(k = pai.nro_chaves-1; k >= i; k--){  
        pai.chaves[k+1] = pai.chaves[k];  
    }  
    pai.chaves[i] = ant.chaves[GRAU-1];  
    pai.nro_chaves++;  
    for(k = pai.nro_chaves; k >= i+1; k--){  
        pai.apontadores[k+1] = pai.apontadores[k];  
    }  
    pai.apontadores[k+1] = novo;  
}
```

Código 13.7

```
public static void inserir_B(NoB a, int ch){
    NoB aux, novo;

    aux = a;

    if(aux.nro_chaves == 2*GRAU-1){
        novo = new NoB();
        novo.folha = 0;
        novo.nro_chaves = 0;
        novo.apontadores[0] = aux;
        a = novo;
        repartir(novo,0);
        inserir_chave_B(aux,ch);
    } else {
        inserir_chave_B(a,ch);
    }
}
```

Código 13.8

```
public static void inserir_chave_B(NoB n, int ch) {
    NoB aux;
    int k;

    k = n.nro_chaves-1;
    if(n.folha == 1){
        while(k >= 0 || ch < n.chaves[k]){
            n.chaves[k+1] = n.chaves[k];
            k--;
        }
        n.chaves[k+1] = ch;
        n.nro_chaves++;
    } else {
        while (k >= 0 || ch < n.chaves[k])
            k--;
        k++;
        aux = n.apontadores[k];
        if(aux.nro_chaves == 2*GRAU-1){
            repartir(n,k);
            if (ch > n.chaves[k])
                k++;
            aux = n.apontadores[k];
        }
        inserir_chave_B(aux,ch);
    }
}
```

Phyton

Código 13.1

```
def alturaArvore(self):
    print altura(self.raiz)

def altura(self, no):
    if no == None:
        return -1
    left = altura(self.arvoreEsquerda.raiz)
    right = altura(self.arvoreDireita.raiz)
    if left > right:
        return left + 1
    else:
        return right + 1
```

Código 13.2

```
def rotacao_direita():
    if self.raiz <> None:
        aux = arvoreEsquerda
        arvoreEsquerda = arvoreDireita
        arvoreDireita.raiz = self.raiz
        self.raiz = aux.raiz
```

Código 13.3

```
def rotacao_esquerda():
    if self.raiz <> None:
        aux = arvoreDireita
        arvoreDireita = arvoreEsquerda
        arvoreEsquerda.raiz = self.raiz
        self.raiz = aux.raiz
```

Código 13.4

```
class NoB(object):
    def __init__(self, grau):
        apontadores = [None]*(2*grau)
        chaves = [None]*(2*grau-1)
        self.grau=grau
```

Código 13.5

```
def criar_arvore_B(self, no):
    novo = NoB()
    novo.folha = 1
    novo.nro_chaves = 0
    no = novo
```

Código 13.6

```
def repartir(self, pai, i):
    ant = NoB()
    novo = NoB()
    k=0
    ant = pai.apontadores[i]
    novo.folha=ant.folha
    while k<= self.grau-2:
        novo.chaves[k] = ant.chaves[k+self.grau]
        k+=1
    if ant.folha == 0:
        k=0
        while k <= self.grau-1:
            novo.apontadores[k] = ant.apontadores[k+self.grau]
            k+=1
    novo.nro_chaves = ant.nro_chaves = self.grau-1
    k=pai.nro_chaves-1
    while k>= i:
        pai.chaves[k+1] = pai.chaves[k]
        k-=1
    pai.chaves[i] = ant.chaves[self.grau-1]
    pai.nro_chaves+=1
    k = pai.nro_chaves
    while k >= i+1:
        pai.apontadores[k+1] = pai.apontadores[k]
        k-=1
    pai.apontadores[k+1] = novo
```

Código 13.7

```
def inserir_B(self, a, ch):
    aux = NoB()
    novo = NoB()

    aux = a

    if aux.nro_chaves == 2*self.grau-1:
        novo = NoB()
        novo.folha = 0
        novo.nro_chaves = 0
        novo.apontadores[0] = aux
        a = novo
        repartir(novo, 0)
        inserir_chave_B(aux, ch)
    else:
        inserir_chave_B(a, ch)
```


Código 13.8

```
def inserir_chave_B(self, n, ch):
    aux = NoB()
    k=0

    k = n.nro_chaves-1
    if n.folha == 1:
        while k >= 0 or ch < n.chaves[k]:
            n.chaves[k+1] = n.chaves[k]
            k-=1
        n.chaves[k+1] = ch
        n.nro_chaves+=1
    else:
        while k >= 0 or ch < n.chaves[k]:
            k-=1
        k+=1
        aux = n.apontadores[k]
        if aux.nro_chaves == 2*self.grau-1:
            repartir(n,k)
            if ch > n.chaves[k]:
                k+=1
            aux = n.apontadores[k]
        inserir_chave_B(aux,ch)
```



Para fixar!

1. Procure destacar as vantagens e as desvantagens que você identificou no uso de árvores balanceadas. Caso encontre dificuldade, utilize simuladores, encontrados na internet, para árvores binárias e árvores AVL.
2. Defina um conjunto de dados e armazene esses dados em duas árvores: uma árvore AVL e uma árvore B. Procure traçar uma comparação entre ambas, destacando principalmente a questão de manutenção do balanceamento na inserção e na remoção de chaves.

Vamos lá, você consegue!



Para saber mais

No caso de interesse por maior aprofundamento nos estudos sobre árvores balanceadas, recomendamos a leitura do Capítulo 18 do livro *Algoritmos: teoria e prática* (Cormen, 2012).



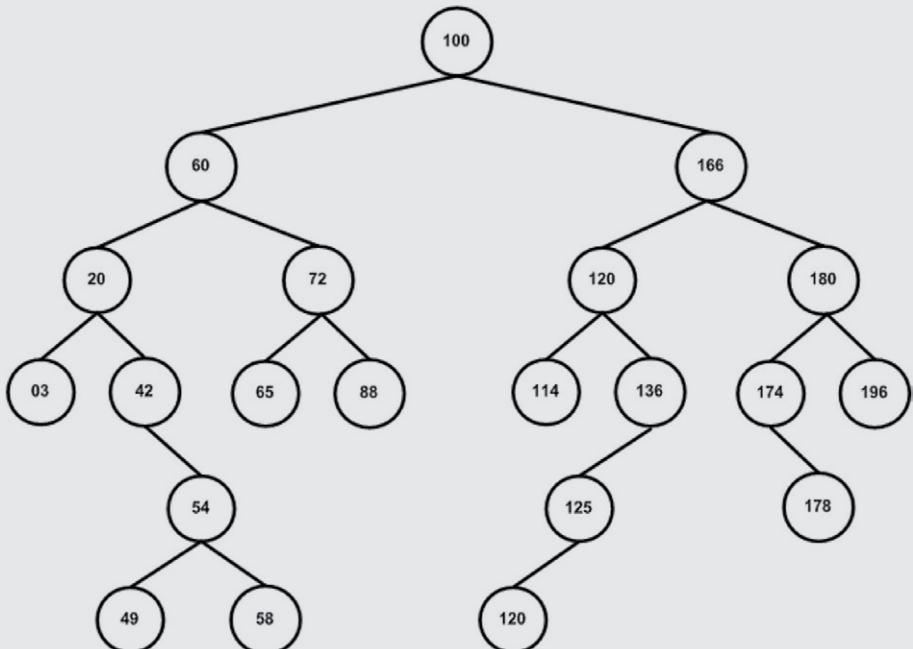
Navegar é preciso

Existem diversas ferramentas disponíveis na internet que podem contribuir para seu aprendizado sobre árvores AVL e árvores B, entre elas:

- Árvore AVL – *link* com código completo na Linguagem C em);
(<http://www.inf.ufrgs.br/~cagmachado/INF01124/t2.htm>)
- Árvore B – *link* com código em Common Lisp em:
(<http://www.sourcecodeonline.com/details/cl-btree.html>);
Applets
(<http://www.cp.eng.chula.ac.th/~vishnu/datastructure/AVL/AVL-Applet.html>),
(<http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>),
(<http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html>),
(<http://slady.net/java/bt/view.php>).

Exercícios

1. Para a árvore binária a seguir, verifique a possibilidade de se aplicar rotações à direita e rotações à esquerda, de forma que ela se torne uma árvore AVL.



2. Procure justificar o motivo de uma árvore AVL crescer “de cima para baixo” e de uma árvore B crescer “de baixo para cima”.
3. Considerando que quanto maior for o número de chaves nos nós folhas numa árvore B, mais simples será a remoção destes, crie uma função que devolva, em determinado momento, o percentual de chaves armazenadas em nós folhas em relação ao número total de chaves armazenadas na árvore.
4. Imagine uma situação em que não mais exista um campo no registro do nó de uma árvore B que indique se esse nó é folha. Com base nessa ideia, crie uma função de devolve quais são os nós folhas e quais são os nós internos de uma árvore B.

Conseguiu? Muito bom!!!

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 2004.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Com este capítulo, “fechamos” os tópicos relacionados à estrutura de dados árvore. Agora você já conhece os conceitos de árvores em geral, de árvores binárias, de árvores N -árias e de árvores balanceadas (árvores AVL e árvores B).

Uma estrutura de dados árvore é um tipo específico de grafo, portanto, com o intuito de ampliar os conceitos, é importante estudar a estrutura e o funcionamento de um grafo. Mas esse é um assunto que ficará para o Capítulo 14.

Grafos

“ *Todos os caminhos estão errados quando você não sabe aonde quer chegar.* ”

WILLIAM SHAKESPEARE

Escolher o melhor caminho, entre muitos, pode significar o alcance rápido de um objetivo. Um grafo pode representar diversos caminhos, todavia, a escolha do melhor depende do algoritmo que projetamos para percorrê-lo.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- identificar um grafo;
- conhecer as principais terminologias de grafos;
- entender os conceitos e métodos para percurso em grafos;
- elaborar e implementar algoritmos para busca ou percurso em grafos.



Para começar



Escolher o melhor caminho dentre vários é uma situação que você precisa enfrentar no cotidiano. Imagine que está planejando uma viagem e, por questões de economia de combustível ou mesmo de redução do tempo dessa viagem, tem de escolher qual é o melhor caminho a ser percorrido para chegar ao destino pretendido.

E então, como você faz isso? Certamente, consciente ou inconscientemente, você constrói um algoritmo para resolver a questão.

Mas não são somente pessoas que, em determinados momentos, têm que decidir qual é o melhor caminho a seguir. Computadores e máquinas também precisam agir dessa forma.

Um exemplo disso são os dispositivos denominados *roteadores (routers)*, utilizados em redes de computadores para encaminhar pacotes de dados entre computador de origem e destino. Para isso, os roteadores baseiam-se em tabelas internas de endereços e rotas de redes a fim de determinar o melhor caminho para o envio de pacotes. Essa atividade do roteador é executada pelos protocolos de roteamento que constituem a porção lógica desse *hardware*.



Papo técnico

Protocolos são algoritmos que atuam nas diversas camadas de uma rede de computadores. Algoritmos de roteamento são executados na camada de rede, segundo o modelo de referência OSI/RM.

A Figura 14.1 mostra um exemplo de redes interligadas por roteadores. Analisando-a, você poderia perguntar: qual seria o melhor caminho para um computador que está ligado no roteador A enviar um pacote de dados para um computador que está ligado no roteador E?

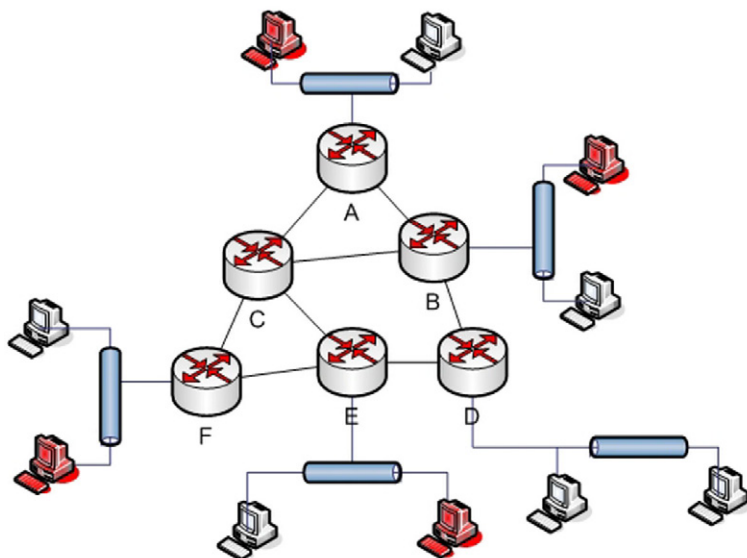


FIGURA 14.1: Protocolos de roteamento estabelecem árvores para conectar todos os clientes de um grupo.

Você sabia que essa e outras soluções são dadas pelos algoritmos, aplicando-se os conceitos sobre grafos, que vamos aprender neste capítulo?

Atenção

Você sabia que “As sete pontes de Königsberg” é um histórico problema da matemática resolvido por Leonhard Euler, em 1736, e que sua solução originou a teoria dos grafos? Pesquise mais sobre esse assunto no site da Wikipedia (www.wikipedia.org).



Então, vamos em frente, que o assunto *grafos* é interessante, abrangente e complexo.

Os grafos são objetos de estudo principalmente da matemática. Os algoritmos de grafos já eram estudados muito antes da construção dos computadores. Muitas pesquisas, que motivaram novas teorias sobre grafos, foram desenvolvidas e concretizadas nos Bell Laboratories e aplicadas, principalmente, na comunicação telefônica em grandes redes. Atualmente,

algoritmos baseados nos conceitos e teorias sobre grafos são largamente aplicados em aplicativos e sistemas (*softwares*) que atuam na internet.

Da mesma forma, algoritmos de grafos são aplicados na construção de processadores e outros componentes eletrônicos para, dentre outras finalidades, determinar a melhor localização de componentes dentro de um *chip*. Na área da bioinformática, temos aplicações de grafos para sequenciamento de cadeia de DNA em projetos de genoma. Portanto, como você pode perceber, o assunto *grafos* é aplicado em larga escala na área da Tecnologia da Informação, tanto em *software* quanto em *hardware*, assim como em outras muitas áreas do conhecimento.



Atenção

Grafos, em ciência da computação, são aplicados na solução de problemas complexos, principalmente os relacionados com inteligência artificial (IA).

Portanto, estudar grafos e elaborar algoritmos para sua implementação significa envolver-se com a complexidade dos algoritmos.

Certamente você conhece o Facebook® ou é um de seus usuários. Então, pense em como as teorias sobre grafos poderiam ser aplicadas em softwares nesse ambiente.

Você sabe que o Facebook® é uma importante rede social em que, dentre outras funcionalidades, podemos estabelecer relações de amizade com as pessoas que integram essa rede, criando grupos de interesses. Um grafo pode, por exemplo, ser construído para mostrar quem se relaciona com determinada pessoa, ou quais grupos se relacionam com determinado grupo.

No “Para começar” do Capítulo 11, você tinha como tarefa identificar o José da Silva e Souza em duas imagens. Na primeira imagem, não foi difícil localizá-lo, mas na segunda foi preciso estabelecer uma organização (hierarquia) na multidão para realizar essa tarefa em um tempo aceitável.

A necessidade de uma hierarquia é uma das limitações das árvores, ou seja, elas não podem representar estruturas de dados em que um item de dado tenha mais de um pai. Os grafos superam essa limitação.



Atenção

Embora um grafo seja uma estrutura de dados muito parecida com uma árvore, ao contrário desta ele não apresenta hierarquia.

Para entender esse conceito, imagine que você esteja em São Paulo e gostaria de participar da conhecida “Festa do Peão” da cidade de Barretos, no interior do estado de São Paulo. Imagine também que, para chegar até essa cidade, você conta com várias opções de caminhos, ou seja, pode seguir por várias estradas, passando por muitas outras cidades.



Dica

Planeje uma rota para tornar essa viagem rápida e, consequentemente, menos cansativa.

Conseguiu planejar a viagem? Se sim, como você fez isso?

Certamente, você estabeleceu alguns critérios para a escolha do melhor caminho. Talvez tenha consultado uma mapa rodoviário do estado de São Paulo e optado por viajar pela melhor rodovia, ou tenha preferido viajar passando pelas maiores cidades do estado. De qualquer forma, você deve ter feito um traçado ligando vários pontos (cidades) por onde passar até chegar em Barretos, e, para tornar a viagem mais rápida, deve ter observado a menor distância entre os pontos traçados.

Se fez isso, então você traçou um grafo (mapa) e observou que ele não possui hierarquia, como as árvores. Uma vez traçado esse grafo, você pode determinar a distância (em quilômetros) entre um ponto e outro, e, então, determinar o caminho mais curto para chegar ao destino pretendido, que é a cidade de Barretos.

Esse grafo pode ter o formato mostrado nas [Figuras 14.2\(1\)](#) - grafo não orientado, aquele que não aponta (seta) para uma direção - ou [14.2\(2\)](#)- grafo orientado, representado por segmentos de reta com setas apontando para uma direção. Mais adiante, veremos os conceitos sobre *grafos orientados* e *grafos não orientados*.



Atenção

Encontrar caminhos entre pontos (nós ou vértices) é um bom argumento para o aprofundamento dos estudos sobre grafos. Pense nisso!

Até este momento, você já aprendeu alguns dos conceitos básicos sobre grafos e sua importância para a ciência da computação.

Nos tópicos a seguir vamos, inicialmente, aprender a definir grafos, conhecer algumas de suas terminologias e também algumas formas de

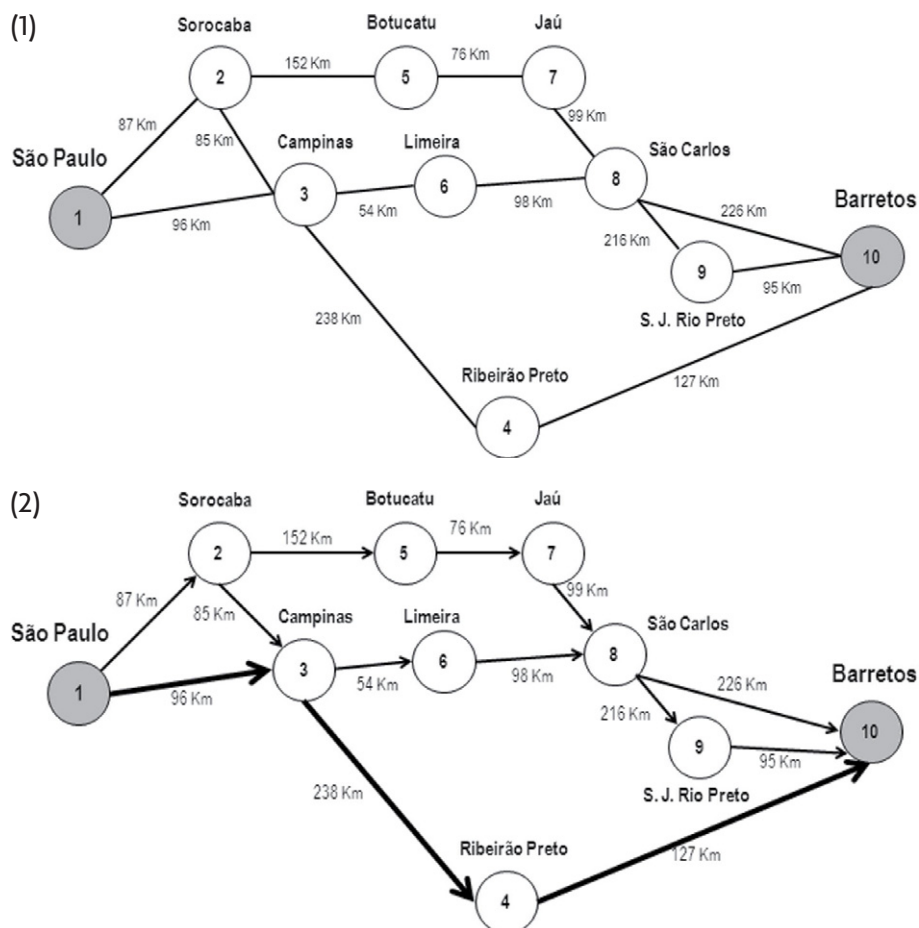


FIGURA 14.2: Exemplo de grafos representando distâncias entre cidades do estado de São Paulo.

representá-los graficamente também como estruturas de dados. Depois, vamos entender os conceitos sobre caminhos em grafos e conhecer dois métodos básicos para percorrê-los, para o que precisaremos construir algoritmos.

Vamos em frente!



Conhecendo a teoria para programar

Definição de grafo

Em termos conceituais, um *grafo* pode ser definido como uma estrutura de dados formada por um conjunto de pontos (nós ou vértices) e um conjunto de linhas (arestas ou arcos) que conectam vários pontos, isto é, estabelecem

uma relação binária entre pares de nós. Portanto, um grafo pode ser assim representado: $G = (N, A)$, onde:

- N representa um conjunto finito de nós ou vértices;
- A representa as arestas ou arcos (relação binária existente entre os nós). As arestas representam os caminhos ou as ligações entre os nós (N_i, N_j) , em que N_i e N_j são elementos de G .

Portanto, $A \leq N \times N$ (o número de arestas deve ser menor ou igual ao produto cartesiano de N por N).

Atenção

O conjunto de nós e de arestas (relações entre nós) devem ser finitos, e um deles pode ser vazio. Portanto, se o conjunto de nós for vazio, então o conjunto de arestas (relações) também será vazio.

Dica

Para entender os conceitos sobre grafos, vamos considerar a seguinte notação: G - referencia um grafo; N - são nós de G ; e A - indica suas arestas.

Consideremos como exemplos um conjunto de nós e outro de arestas, que definem um grafo G_1 com cinco nós, rotulados de 1 a 5, e quatro relações binárias entre eles. Temos, então:

$G_1 = (N, A)$, onde:

$N = \{1, 2, 3, 4, 5\}$ e $A = \{(1, 2), (1, 4), (3, 5), (4, 5)\}$

Conforme mostrado na [Figura 14.3](#), cada aresta do conjunto de relações (A) constitui um conjunto de dois nós. Temos, então, uma aresta entre 1 e

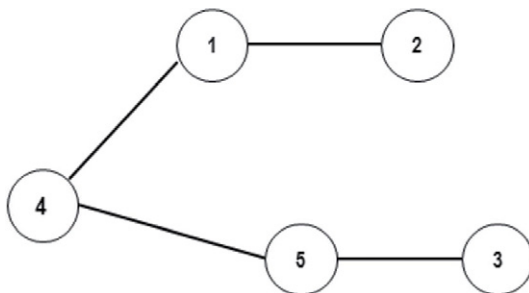


FIGURA 14.3: Representação de um grafo.

2, 1 e 4, 3 e 5, e 4 e 5. Se temos uma aresta entre qualquer par de nós (x,y) , então existe uma passagem (caminho) entre o nó x e o nó y , e vice-versa.

Formas de representar grafos

Representação gráfica de grafos

Para a representação gráfica de um grafo, usamos a seguinte simbologia:

- círculo ou ponto para representar um nó (N);
- um segmento de reta para representar uma aresta (A) - ligação ou caminho entre os nós (em grafo não orientado ou não direcionado), conforme mostrado na [Figura 14.2\(1\)](#);
- um segmento de reta *com uma seta* em uma das extremidades, para representar uma aresta (A) - ligação ou caminho entre os nós, ou caminho de volta para o próprio nó (em grafo orientado ou direcionado), conforme mostrado na [Figura 14.2\(2\)](#).

Atenção



Fisicamente, podemos desenhar o mesmo grafo de várias formas. Portanto, a representação física dos nós e até de seus rótulos (identificação) não tem relevância. O importante é manter íntegro o relacionamento entre os nós, qualquer que seja sua disposição física.

A [Figura 14.4](#) mostra o grafo da [Figura 14.3](#) disposto fisicamente de outra forma.

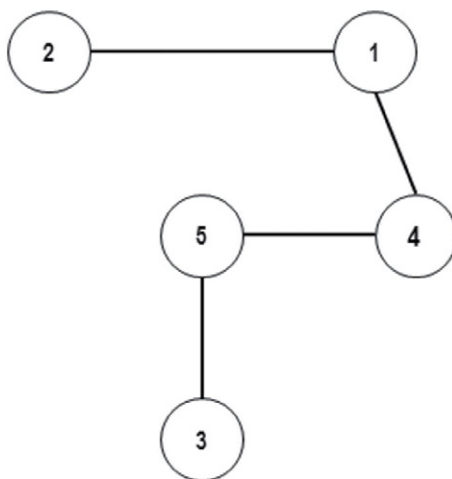


FIGURA 14.4: Outra forma de representação de grafo.

A forma de representação gráfica de grafos, que acabamos de estudar, é a mais utilizada na literatura sobre o assunto. Todavia, dependendo do problema a ser tratado por meio de grafos, podem existir outras formas de representá-lo para mostrar a solução.

A Figura 14.5 mostra exemplos de grafos de problemas reais. Na Figura 14.5 (1), temos um circuito de polarização de um transistor, que pode ser visto como um grafo. Na Figura 14.5(2), temos um grafo representando uma máquina de estados. Por fim, a Figura 14.5(3) mostra o processo de fabricação de cerveja, que também pode ser visto como um grafo.

Terminologia de grafos

Grafo orientado ou direcionado

Um grafo é *orientado* se suas arestas são orientadas (arcos), porém a existência de uma aresta de um nó x para y não garante que exista um caminho nas duas direções.



Papo técnico

Um grafo que contém arestas orientadas (arcos) também é referenciado como um dígrafo. A palavra *dígrafo* tem origem no termo inglês *digraph*, que é a contração da expressão *directed graph*, que significa “grafo dirigido ou orientado”.

Na Figura 14.6, temos um grafo orientado, em que se percebe o uso de segmentos de reta com setas para representar a ligação entre os nós, e, no caso do nó 2, a ligação consigo mesmo.

Para entender os conceitos de grafo orientado, vamos considerá-lo $G_1(N,A)$, em que:

- $N = \{1,2,3,4,5\}$ representa o conjunto finito, não vazio, de nós ou vértices de G_1 ;
- $A = \{(1,4),(1,2),(2,2),(2,5),(3,5),(4,1),(4,5),(5,3)\}$ representa o conjunto finito, não vazio, de pares ordenados de nós ou vértices de G_1 ;

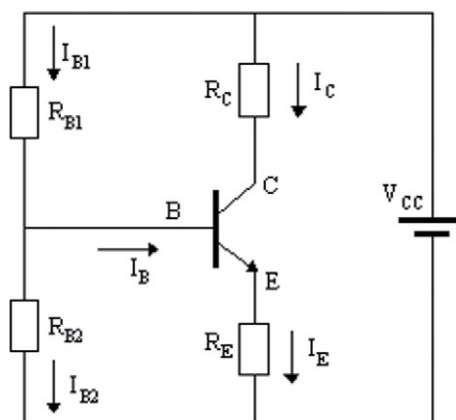


Atenção

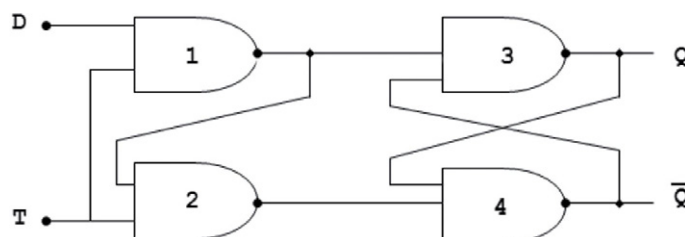
Um par é ordenado quando suas arestas são distintas.

- um arco ou aresta (s,t) que pertence ao subconjunto $A - (s,t \in A)$ - é representada por $s \rightarrow t$;
- um arco que aponte de s para t é chamado de *arco direcionado*.

(1) Circuito de polarização com divisor de tensão de um transistor



(2) Diagrama de estados



(3) Processos de fabricação industrial de cerveja

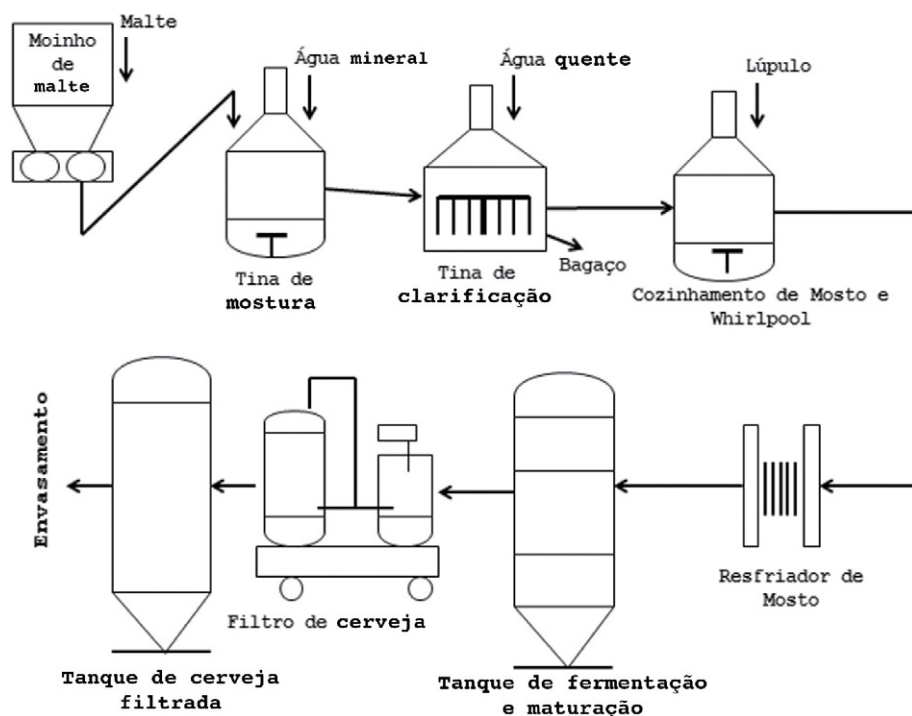


FIGURA 14.5: Exemplos de grafos representando situações reais.

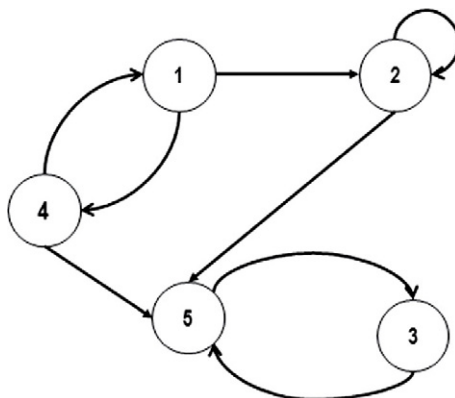


FIGURA 14.6: Representação de um grafo orientado.

Relação de adjacência de um nó

- O nó s é chamado de *origem do arco*, pois está no início dele, enquanto o nó t é chamado de *destino do arco*. Portanto, s e t são nós adjacentes.

Considerando a [Figura 14.6](#), temos que o nó 1 é adjacente ao nó 4, e, por sua vez, o nó 4 é adjacente ao nó 1; já o nó 2 é adjacente ao nó 1 e a si mesmo.



Conceito

- Dois nós são *adjacentes* quando estão conectados um ao outro por uma única aresta ou arco.
- Um nó é *adjacente a um outro* se existe uma aresta chegando a ele partindo do outro nó.

Grau de um nó

É a quantidade de arestas ou arcos incidentes em um nó (que chegam ao nó e partem dele):

- considerando a representação (s,t) , pode-se considerar “parte do nó s para t ”

A notação $P(s)$ representa o conjunto de arestas ou arcos que partem de um nó, no caso do nó s . Portanto:

$$P(s) = \{(s_0, s_1) \in A : s_0 = s\}$$

- considerando a representação (s,t) , pode-se considerar “chega ao nó t partindo de s ”.

A notação $I(s)$ representa o conjunto de arestas ou arcos que chegam até um Nó, no caso do Nó t . Portanto:

$$I(t) = \{(s_0, s_1) \in A : s_1 = t\}$$



Atenção

Em grafo orientado são permitidos laços, isto é, “parte de um nó e chega ao mesmo nó”.

Grau de saída de um nó

- O número ou grau de saída (grau externo) de um nó é o número de arestas ou arcos que partem desse nó.

O grau de saída de s é $|P(s)|$

Considerando a [Figura 14.7](#), temos que o nó 1 é de grau de saída 1 (apresenta uma saída), enquanto o nó 2 é de grau de saída 3 (apresenta 3 saídas).

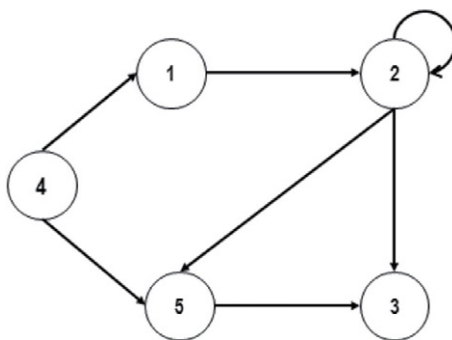


FIGURA 14.7: Representação de um grafo orientado.

- O número ou grau de entrada (grau interno) de um nó é o número de arestas ou arcos que incidem sobre esse nó.

O grau de entrada de t é $|A(t)|$

Considerando a [Figura 14.7](#), temos que o nó 1 é de grau de entrada 1 (apresenta 1 entrada), enquanto o nó 2 é de grau de entrada 2 (apresenta 2 entradas), pois esse nó possui um laço.



Atenção

O grau de um grafo = (grau de saída + grau de entrada) - de todos os nós.

Grafo não orientado ou não direcionado

Um grafo é considerado não orientado ou não direcionado quando os nós ou vértices estão conectados por arestas ou arcos não direcionados, sendo representados por segmento de reta. Portanto, as duas extremidades da aresta são equivalentes (não apresentam flechas) e não existem origem e destino.



Atenção

Em um grafo não orientado, uma aresta é representada como um conjunto, e não como um par ordenado, como ocorre no grafo orientado.

Para entender o conceito de grafo não orientado, vamos considerar o grafo da [Figura 14.8](#), que contém quatro nós e quatro arestas. Temos, então, um grafo $G_2(N,A)$, onde:

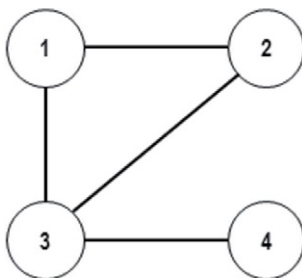


FIGURA 14.8: Representação de um grafo não orientado.

- $N = \{1,2,3,4\}$ representa um conjunto finito, não vazio, de nós ou vértices de G_2 ;
- $A = \{\{1,2\},\{1,3\},\{2,3\},\{3,4\}\}$ representa um conjunto finito de conjuntos de arestas ou arcos de G_2 .

Observe o grafo da [Figura 14.8](#).

Você concorda que uma aresta em um grafo não orientado é um conjunto $\{1,2\}$ $\{2,1\}$ (congruentes) e que A também é um conjunto de arestas?

Então, considerando isso, percebe-se que, em um grafo não orientado, o conjunto de arestas (A) não pode conter mais de uma instância da mesma aresta (por exemplo: $\{1,2\}$ e $\{2,1\}$).



Atenção

Em um grafo não orientado não pode existir laços, porque uma aresta é um conjunto de comprimento dois e, portanto, em um conjunto não pode haver duplicidades.

Grafos rotulados

Um grafo rotulado é aquele que apresenta informações adicionais em nós ou vértices, em arestas ou arcos, ou em ambos, como é o caso do grafo da [Figura 14.2](#). A [Figura 14.5\(3\)](#) apresenta os nós rotulados. Supondo que o rótulo fosse um número de processo, certamente existiria uma tabela de equivalência, anexa ao grafo, indicando qual é o processo ocorrendo em cada nó.

Muitas aplicações práticas de grafos exigem que estes sejam rotulados.

Caminhos e ciclos em grafos

Conforme já foi dito, encontrar caminhos entre pontos (nós ou vértices) em um mapa, circuito, rede, etc. é uma boa razão para estudarmos grafos. Então, vamos conhecer algumas terminologias de grafos referente a caminhos:

- **caminho em grafo** - sequência de nós ou vértices em que cada nó sucessivo é adjacente ao seu predecessor. Na [Figura 14.9](#), a sequência de nós São Paulo-Campinas-Ribeirão Preto-Barretos indica um caminho;

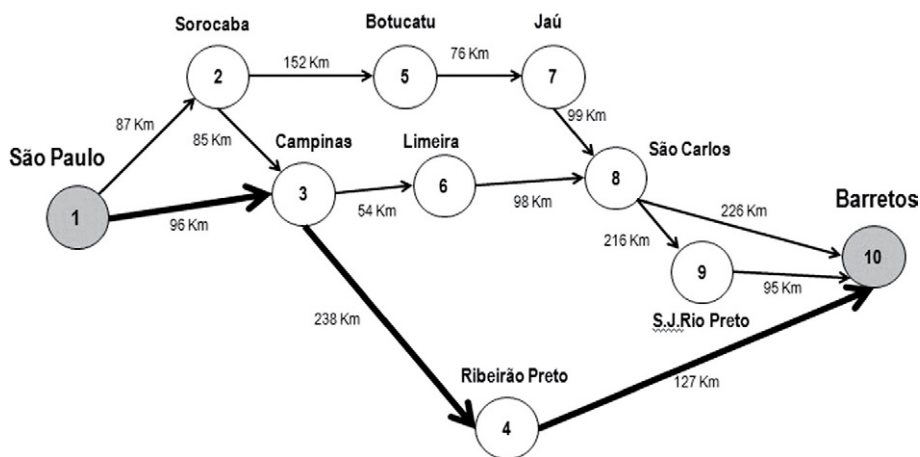


FIGURA 14.9: Grafo indicando o caminho da cidade de São Paulo até a cidade de Barretos.

- **caminho simples** - quando a sequência de nós ou vértices é distinta, exceto pelo fato de o primeiro e o último nó poderem ser o mesmo, ou seja, o mesmo caminho de ida será o de volta. Portanto, o caminho de São Paulo a Barretos, conforme mostrado na Figura 14.9, é um *caminho simples*.

Agora observe nessa figura o caminho Barretos-São Paulo, passando por São Carlos, Jaú, Botucatu e Sorocaba. Esse *não* é um caminho simples. Você sabe por quê?

Porque nesse caminho, partindo de Barretos, temos a opção de ir por São José do Rio Preto, portanto, o caminho de volta pode não ser o mesmo. Veja a seguir:

- **caminho cíclico** - é quando existe um caminho de no mínimo três nós que começa e termina no mesmo nó, ou seja, é um caminho simples em que apenas o primeiro e o último nós são iguais. A Figura 14.10 mostra um caminho cíclico, Barretos-José do Rio Preto-São Carlos-Barretos.
- **caminho acíclico** - quando não ocorre nenhum ciclo no caminho em circuito (caminho) nos nós. Observe na Figura 14.10 o caminho Sorocaba-Campinas-Ribeirão Preto. É um caminho simples e acíclico.

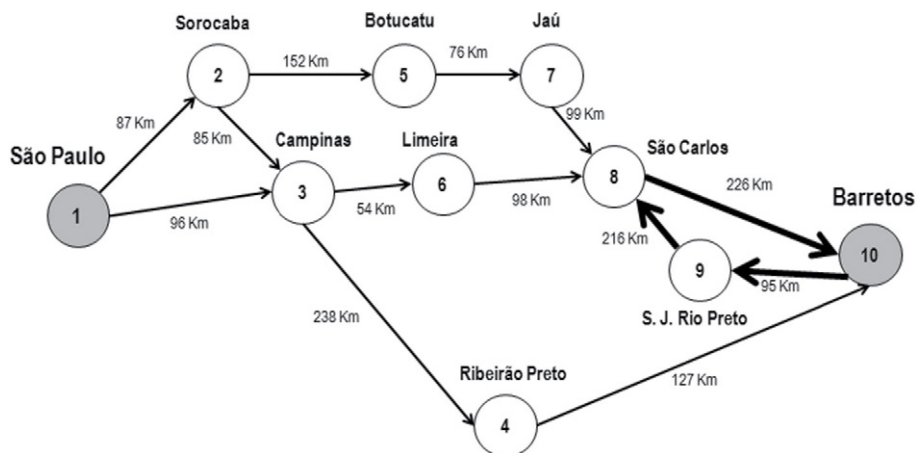


FIGURA 14.10: Grafo cíclico partindo da cidade de Barretos e retornando a ela.

Representação de grafos como estruturas de dados

Para representar um grafo, precisamos de no mínimo duas estruturas de dados: uma para armazenar os nós ou vértices e outra para armazenar as arestas ou arcos, ou seja, os dois conjuntos que formam um grafo - $G(N,A)$.



Dica

A partir de agora, vamos aplicar os conceitos estudados em capítulos anteriores. Portanto, se tiver dúvidas, você sabe onde procurar as informações para saná-las.

Que tal aprendermos um pouco mais sobre a aplicação de estruturas de dados?

As estruturas de dados que podemos usar para representar grafos e armazenar seu conteúdo são apresentadas a seguir.

Lista de adjacência

Uma das formas mais flexíveis de representar um grafo é a lista de adjacências, que utiliza um vetor com n listas ligadas.

Para a construção de uma lista de adjacências, consideremos como exemplo o grafo orientado da [Figura 14.11](#) - $G(N,A)$.

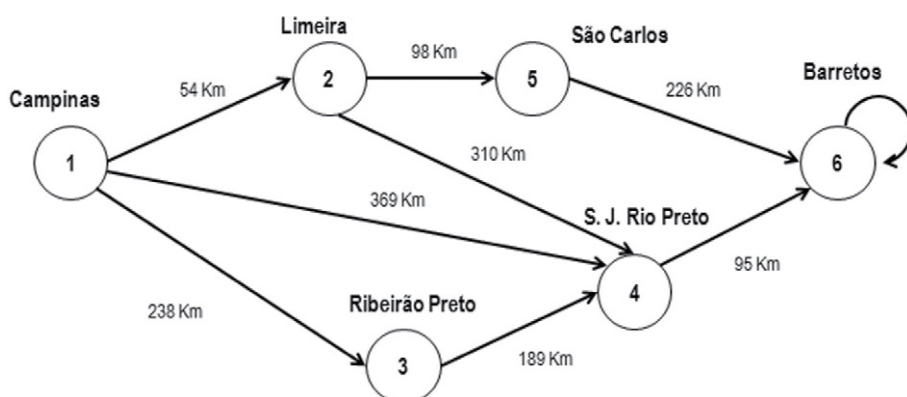


FIGURA 14.11: Grafo orientado.

A lista de adjacência mostrada na [Figura 14.12](#) foi construída a partir do grafo G da [Figura 14.11](#).

A lista de adjacências para o grafo G acima utiliza um vetor (nós de G) com 3 listas ligadas, em que a cada nó se associa a lista dos nós que lhe são adjacentes. Cada posição (elemento) do vetor corresponde a um nó de $G(N,A)$, e as arestas de um certo nó correspondem a outros nós.

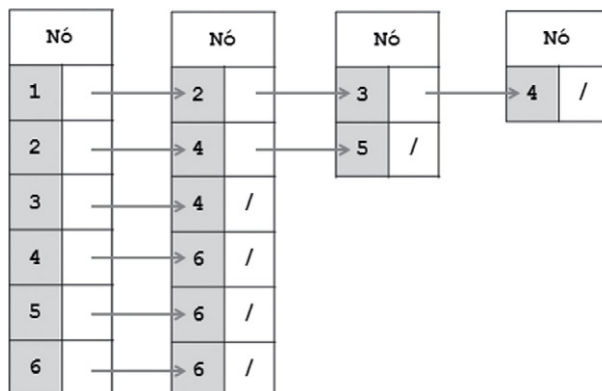


FIGURA 14.12: Lista de adjacência.



Papo técnico

Para a remodelagem de um grafo em tempo de execução de um programa, é imprescindível o uso da alocação dinâmica de sua representação (*malloc*). Portanto, a representação de adjacências entre os nós pode ser feita por meio de listas lineares. De cada elemento índice parte uma lista encadeada mostrando os nós adjacentes conectados.

Matriz de adjacência

Para representar um grafo por meio de uma matriz de adjacências, utiliza-se uma matriz bidimensional (matriz $N \times N$, ou matriz linha $\times$ coluna). Nessa matriz, assume-se que os nós são numerados de 1 até n , nas linhas e nas colunas, sendo que os nós origem estão representados nas linhas, e os nós destinos nas colunas.

Temos, então, uma matriz $M[i,j]$, onde i e j são os nós do grafo.

Para representar a existência ou a ausência de arestas ou arcos entre os nós, pode-se simplesmente atribuir valor numérico 1 (existe uma aresta) ou 0 (não existe uma aresta), ou um valor booleano - *True* ou *False*, quando o grafo é não valorado.



Atenção

Um grafo é denominado valorado quando possui peso (valor) em suas arestas. O grafo da [Figura 14.11](#) é um grafo valorado.

Portanto, $M[i,j] = 1$ ou *True* se i é adjacente a j , ou 0 ou *False*, em caso contrário.

A matriz de adjacência, mostrada na [Figura 14.13](#), foi construída a partir do grafo G da [Figura 14.11](#).

		Nós Destino					
		1	2	3	4	5	6
Nós Origem	1	0	1	1	1	0	0
	2	0	0	0	1	1	0
	3	0	0	0	1	0	0
	4	0	0	0	0	0	1
	5	0	0	0	0	0	1
	6	0	0	0	0	0	1

FIGURA 14.13: Matriz de adjacência.

Matriz de incidência

A matriz de incidência é uma forma muito utilizada para representação de grafos. Nessa matriz, as linhas representam os nós do grafo, e as colunas representam as arestas ou arcos, que devem ter um peso ou valor para sua representação.

```
int MatAdj [6][6]; // Definição, em linguagem C, da matriz de adjacência do exemplo anterior.
```

Para a construção de uma matriz de incidência, consideremos como exemplo o grafo orientado com arestas valoradas da [Figura 14.14](#) - $G(N,A)$.

Para a construção de uma matriz de incidência, a partir de um grafo orientado e valorado, é importante distinguir nó origem e nó destino das arestas. Portanto, vamos utilizar os números inteiros -1, 1 e 0 para o preenchimento da matriz, sendo que:

- o inteiro -1 denota a origem da aresta (e também um laço);
- o inteiro 1 denota o destino da aresta;
- o 0 representa nós que não participam de uma aresta.

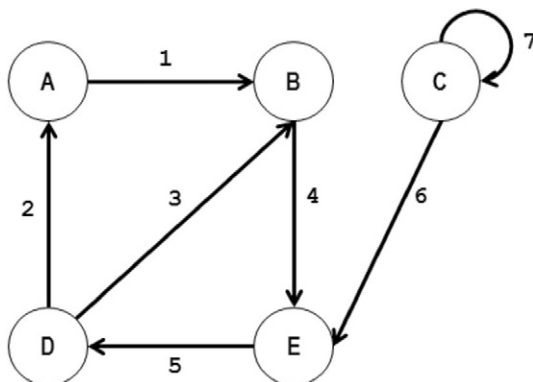


FIGURA 14.14: Grafo orientado com arestas valoradas.

A matriz de incidência, mostrada na [Figura 14.15](#), foi construída a partir do grafo G da [Figura 14.14](#).

		Arestas						
		1	2	3	4	5	6	7
Nós	A	-1	1	0	0	0	0	0
	B	1	0	1	-1	0	0	0
	C	0	0	0	0	0	-1	-1
	D	0	-1	-1	0	1	0	0
	E	0	0	0	1	-1	1	0

FIGURA 14.15: Matriz de incidência.

Métodos para percurso em grafos

Para percorrer um grafo, examinando todos os seus nós e vértices, podemos utilizar alguns procedimentos sistemáticos (ou algoritmos) de percurso ou busca em grafos. Esses procedimentos consistem em visitar cada nó ou vértice em ordem sistemática.

Assim como foi visto no Capítulo 11 sobre árvores, no caso dos grafos podemos escrever algoritmos para realizar buscas ou percursos, utilizando alguns métodos. Para percurso e busca em grafos, existem basicamente dois métodos:

- percurso ou busca em largura (*breadth first search*);
- percurso ou busca em profundidade (*depth first search*).

Esses métodos serão implementados em C, mais adiante.



Atenção

Embora esses métodos façam percursos em grafos, são comumente conhecidos como métodos ou algoritmos de busca em largura e de busca em profundidade.

Método para percurso em largura

Aplicamos esse método quando pretendemos verificar se todos os principais vértices de um grafo estão antes de seu predecessor (ou derivações posteriores). Esse método consiste em percorrer o grafo, passando (visitando) por todos os seus nós.

Como você pode observar, esse método é semelhante ao de percurso em largura de uma árvore binária. Todavia, no caso das árvores, você tem que seguir a hierarquia, ou seja, começar visitando todos os nós de nível 0, em seguida os de nível 1, e assim por diante. No caso dos grafos, como não existe hierarquia, é necessário fazer a *escolha arbitrária* de um “nó de partida”.

Dessa forma, o percurso em largura de grafos visita primeiro o nó inicial escolhido; em seguida, visita todos os seus nós adjacentes, e depois todos os nós adjacentes a esses. O percurso termina quando todos os nós foram visitados.



Dica

Você pode definir o nível de um vértice como o comprimento do menor caminho do vértice inicial até ele. Dessa forma, pode determinar o menor caminho no percurso.

A seguir, temos um algoritmo na forma descritiva, enumerando uma sequência de passos para percorrer grafos em largura.

Algoritmo Percurso em Largura.

Início

P1: Escolha um Nó para início do percurso.

P2: Coloque esse "Nó de partida" em uma fila de Nós "visitados"

P3: Visite os Nós adjacentes ao Nó escolhido.

P4: Coloque (Marque) os Nós adjacentes em um fila de Nós "visitados".

P5: Coloque cada Nó adjacente numa fila de adjacentes.

P6: Após visitados os Nós adjacentes, escolha o primeiro Nó da fila.

P7: Reinicie o algoritmo a partir de P3 até que todos os Nós tenham sido visitados, ou o Nó procurado tenha sido encontrado.

Fim.

Considerando o grafo orientado da [Figura 14.16](#) e aplicando o algoritmo acima, escolhendo o nó 2 como ponto de partida, temos a seguinte ordem de execução:

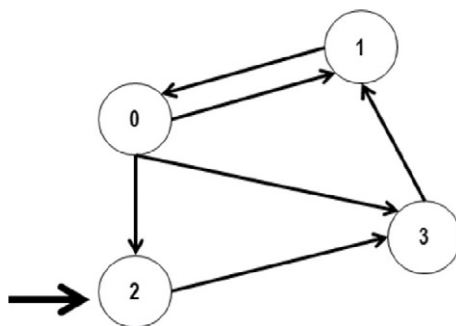


FIGURA 14.16: Grafo orientado.

1. O nó 3 é o primeiro adjacente do Nó 2, então Nó 3 vai para a fila de Nós adjacentes

3	/
---	---

Nós Visitados

2	3
---	---

Como não há outro Nó adjacente ao Nó 2, então escolhe-se o primeiro Nó da fila (3), e reinicia a busca (Passo 3) de Nós adjacentes ao Nó 3.

2. O nó 1 é o primeiro adjacente do Nó 3, então Nó 1 vai para a fila.

1	/
---	---

Nós Visitados

2	3	1
---	---	---

Novamente não há outro Nó adjacente ao Nó 1, então escolhe-se o primeiro Nó da fila (1), e reinicia a busca (Passo 3) de Nós adjacentes ao Nó 1.

3. O nó 0 é o primeiro adjacente do Nó 1, então Nó 0 vai para a fila.

0	/
---	---

Nós Visitados

2	3	1	0
---	---	---	---

A ordem do percurso a partir do Nó 2 é:

2	3	1	0
---	---	---	---

Código 14.1

```
// Procedimento de percurso ou busca em Largura, em grafos - (BFS - breadth first search)

#include <stdio.h>           // Biblioteca com operações de I/O (Input/Output)
#include <string.h>          // Biblioteca com operações sobre String (cadeia de caracteres)
#include <stdlib.h>          // Biblioteca de propósito geral padrão da linguagem C

#define MAXNOD 6             // Número máximo de Nós do grafo

void busca_largura(int inicio, int fim, int lista_nos[], int vet_dista[],
                  int cardP[], int lista_adj_P[][MAXNOD])

// inicio e fim - nó inicial e final do grafo
// lista_nos - lista contendo os Nós de um Grafo
// vet_dista - lista de arestas do grafo
// cardP - vetor de cardinalidade do grafo
// lista_adj_P - matriz de adjacências
{
    int wk_no, ind_ln, n_ini, n_fim, ind_la, wk_la, indvet;
    n_ini = fim + 1;
    n_fim = fim;

    for (ind_ln = inicio; ind_ln <= fim; ind_ln++)
    {
        wk_no = lista_nos[ind_ln]; // escolhe primeiro nó do grafo para início do caminhamento

        for (ind_la = 1; ind_la <= cardP[wk_no]; ind_la++) // Percorre arestas do Nó escolhido
        {
            wk_la = lista_adj_P[wk_no][ind_la]; // salva posição da aresta

            if (vet_dista[wk_la] == -1) // Verificada se nó selecionado possui aresta
            {
                vet_dista[wk_la] = vet_dista[wk_no] + 1; // visita vértices adjacentes
                n_fim++;
                lista_nos[n_fim] = wk_la; // marca nó visitado
            }
        }
    }

    // Chamada recursiva do procedimento

    if (n_ini <= n_fim)
        busca_largura(n_ini, n_fim, lista_nos, vet_dista, cardP, lista_adj_P);
}
```

Método para percurso em profundidade

Aplicamos esse método quando pretendemos explorar ao máximo os vértices de determinada ramificação (caminho) de um grafo.

Esse método consiste em escolher um nó aleatório num grafo e, em seguida, visitar um de seus nós adjacentes. Aprofundando mais a visita, repete-se esse processo recursivamente ou não até atingir o último nó da ramificação.

A seguir, temos um algoritmo, na forma descritiva, enumerando uma sequência de passos para percorrer grafos em profundidade.

Algoritmo Percurso em Profundidade.

Início

P1: Escolha um Nó n , aleatoriamente, para início do percurso.

P2: Marque esse Nó n como "visitado".

P3: Empilhe esse Nó n em uma pilha v .

P4: Enquanto a pilha v não estiver vazia, faça:

P4.1: Desempilhe o Nó n da pilha v .

P4.2: Para cada Nó m (não marcado) adjacente
ao Nó n , faça:

P4.2.1: Visite o Nó m .

P4.2.2: Coloque o No n na pilha v .

P4.2.3: Marque o Nó m como "visitado".

P4.2.4: Mova Nó m para Nó n .

Fim.

Considerando o grafo orientado da [Figura 14.17](#) e aplicando o algoritmo acima, escolhendo o nó 0, arbitrariamente, como vértice inicial de partida, temos a seguinte ordem de execução:

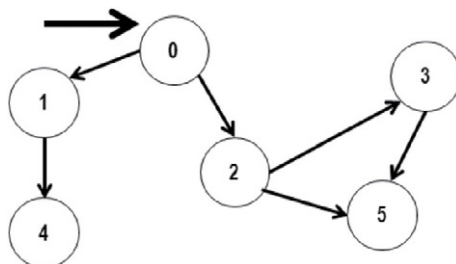


FIGURA 14.17: Grafo orientado.

1. O nó 0 vai para pilha v como o primeiro Nó visitado.

pilha v

0

Nós Visitados

0

Enquanto a pilha v não estiver vazia:

2. desempilha-se o nó n da pilha v ;
3. visita-se um Nó adjacente ao Nó 0, por exemplo o Nó 1. Empilha o Nó 0, e o Nó 1 torna-se o vértice inicial (P4.2.4.);
4. repete-se o processo, visitando o Nó 4 que é adjacente ao Nó 1. Empilha o Nó 1.

pilha v

1
0

Nós Visitados

0	1
---	---

Não havendo mais Nós adjacentes ao Nó 4, então desempilha-se o topo, isto é, o Nó 1. Todavia, como esse Nó não tem adjacentes, então desempilha-se o Nó 0, que tem o Nó 2 como adjacente;

5. empilha-se o Nó 0, e o Nó 2 torna-se o Nó inicial;

pilha v

0

Nós Visitados

0	1	4
---	---	---

6. visita-se um Nó adjacente ao Nó 2, por exemplo o Nó 5. Empilha o Nó 2, e o Nó 5 torna-se o vértice inicial (P4.2.4.);

pilha v

2
0

Nós Visitados

0	2	5
---	---	---

Não havendo mais Nós adjacentes ao Nó 5, então desempilha-se o topo, isto é, o Nó 2, que tem o Nó 3 como adjacente. Empilha-se o Nó 2 novamente, e o Nó 3 torna-se o Nó inicial.

pilha v

2
0

Nós Visitados

0	2	3
---	---	---

7. visita-se um nó adjacente ao nó 3 e verifica-se que esse nó já foi visitado, então, desempilha-se o nó 2 que está no topo. Como esse nó não tem mais nós adjacentes “não visitados”, então desempilha-se o nó 0, que também não tem mais nós adjacentes “não visitados”;
8. a pilha ficou vazia, e o percurso ou busca terminou.

A ordem do percurso em profundidade do grafo da [Figura 14.17](#), tomando como ponto de partida o nó 0 é:

a)

0	1	4
---	---	---

 b)

0	2	5
---	---	---

 c)

0	2	3
---	---	---

O Código 14.2, escrito na linguagem C, implementa o procedimento busca_profundidade para o percurso em profundidade, em um grafo orientado.

Código 14.2

```
// Procedimento de percurso ou busca em Profundidade, em Grafos a partir da origem - (DFS - depth first search)

#include <stdio.h>          // Biblioteca com operações de I/O (Input/Output)
#include <stdlib.h>         // Biblioteca de propósito geral padrão da linguagem C

#define MAXNOD 6           // Número máximo de Nós do grafo

void busca_profundidade(int vertice, int listAdjP[][MAXNOD], int mark[], int cardP[])

//listAdjP - matriz de adjacências do grafo
//mark - vetor com vértices percorridos
//cardP - cardinalidade do grafo (Nós adjacentes)
{
    int w_vertice, ind;
    if (mark[vertice] == 0)    //0 - vértice não percorrido; 1 - vértice percorrido
    {
        printf(" %d\n", vertice);                //Mostra Nó a ser percorrido
        mark[vertice] = 1;                        //Retira o vértice do grafo
        for (ind = 1; ind <= cardP[vertice]; ind++)
        {
            w_vertice = listAdjP[vertice][ind];    //Próximo Nó a ser visitado
            printf(" %d -> %d ", vertice, w_vertice); //Mostra Nós percorridos
            printf("\n");
            busca_profundidade(w_vertice, listAdjP, mark, cardP); //recursividade
        }
    }
}
```

Aplicações de grafos

Caminho euleriano

Um grafo apresenta um caminho euleriano se e somente se não houver nenhum nó ímpar ou se existirem no máximo dois nós ímpares. O caminho euleriano em um grafo visita cada aresta apenas uma vez. Portanto, para ser considerado um grafo euleriano, uma das principais condições, porém não a única, é que todos os nós precisam ser de grau par.

No caso de não haver nós ímpares, o percurso pode se iniciar em qualquer nó e ser finalizado nesse mesmo nó inicial. Ocorrendo dois nós ímpares, o percurso deve se iniciar em um nó ímpar e terminar no outro.

O Código 14.3, a seguir, escrito na linguagem C, implementa o procedimento *caminho_euleriano*, que determina se um grafo possui um caminho euleriano, conforme explicado.

Código 14.3

```
// Procedimento para determinar se um grafo possui um Caminho Euleriano

#define MAXNOO 3          // Numero máximo de Nós Origem no Grafo G
#define MAXNOD 3          // Numero máximo de Nós Destino no Grafo G

void caminho_euleriano(int mat_adj[][MAXNOD], int nos) // Recebe a Matriz de Adjacências do Grafo G e
                                                         // a quantidade de Nós do grafo
{
    int qtd_nos_impar, tot_grau_nos, lin, col, fim;
    qtd_nos_impar = 0;
    lin = 0;
    while ((qtd_nos_impar <= 2) && (lin < nos)) // Quando quantidade Nós Ímpares > 2 interrompe repetição
    {
        tot_grau_nos = 0;
        for (col=0; col < nos; col++)
            tot_grau_nos = tot_grau_nos + mat_adj[lin][col]; // Total de Graus dos Nós

        if (tot_grau_nos%2 != 0) // determina quantidade de Nós Ímpares
            qtd_nos_impar++;
        lin++;
    }
    if (qtd_nos_impar > 2)
        printf("\n\n NAO existe um caminho Euleriano no Grafo G !");
    else
        printf("\n\n EXISTE um caminho Euleriano Grafo G !");
}
```

O Código 14.4, escrito na linguagem C, implementa o procedimento *caminho_euleriano* sobre o grafo orientado da [Figura 14.11](#), usando como entrada do procedimento a matriz de adjacências desse grafo, mostrada na [Figura 14.13](#).

Código 14.4

```
//Determina se existe, ou não, caminho euleriano no Grafo da Figura 14.11

#include <stdio.h>           //Biblioteca com operações de I/O (Input/Output)
#include <string.h>          //Biblioteca com operações sobre String (cadeia de caracteres)
#include <stdlib.h>          //Biblioteca de propósito geral padrão da linguagem C
#define MAXNOO 6            //Numero máximo de Nós Origem
#define MAXNOD 6            //Numero máximo de Nós Destino

void caminho_euleriano(int mat_adj[][MAXNOD], int nos)
{
    int qtd_nos_impar, tot_grau_nos, lin, col;
    qtd_nos_impar = 0;
    lin = 0;

    while ((qtd_nos_impar <= 2) && (lin < nos))
    {
        tot_grau_nos = 0;
        for (col=0; col < nos; col++)
        {
            tot_grau_nos = tot_grau_nos + mat_adj[lin][col];
        }

        if (tot_grau_nos%2 != 0)
            qtd_nos_impar++;
        lin++;
    }

    if (qtd_nos_impar > 2)
        printf("\n\n Não existe um caminho Euleriano!");
    else
        printf("\n\n Existe um caminho Euleriano!");
}

int main()                // função principal
{
    int mat_adjacencias[MAXNOO][MAXNOD] = {{0,1,1,1,0,0},
                                             {0,0,0,1,1,0},
                                             {0,0,0,1,0,0},
                                             {0,0,0,0,0,1},
                                             {0,0,0,0,0,1},
                                             {0,0,0,0,0,1}};

    int fim, lin, col;
    printf("\n Matriz de Adjacencias: \n");
    fim = MAXNOD - 1;
    for (lin = 0; lin <= fim; lin++)
    {
        printf("\n");
        for (col = 0; col <= fim; col++)
            printf(" %d ", mat_adjacencias[lin][col]);
    }
}
```

```

caminho_euleriano(mat_adjacencias, 6); // Chamada do procedimento - Caminho Euleriano

printf("\n");
printf("\n>>>> ");
system("pause"); // system - interrompe execução até ser pressionado ENTER
return 0;
}

```

Caminho hamiltoniano

Se o caminho começa e termina no mesmo vértice, temos um *ciclo hamiltoniano*.

Um caminho hamiltoniano é aquele que contém cada nó do grafo exatamente uma vez, ou seja, o caminho passa por todos os nós sem repetição desses nós e, consequentemente, de arestas. É diferente do caminho euleriano, em que não se repetem arestas, mas podem-se repetir nós. Se o caminho começa e termina no mesmo nó, temos um ciclo hamiltoniano.

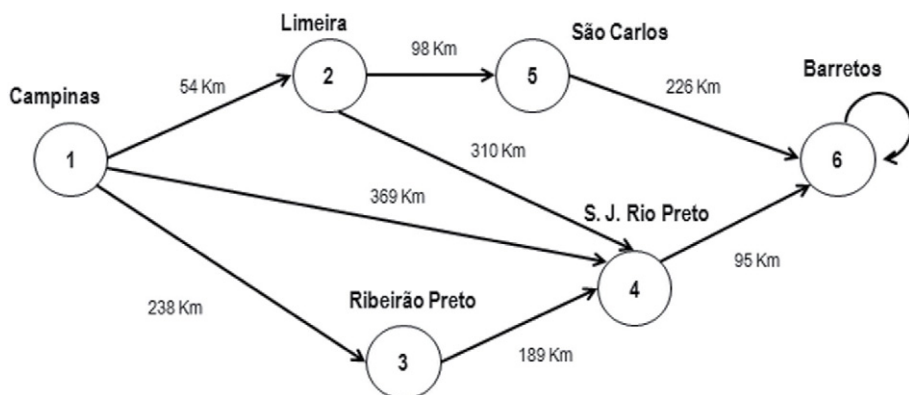
No link: <http://www.geeksforgeeks.org/backtracking-set-7-hamiltonian-cycle/>, você encontrará um programa em C que define a função que determina se existe um caminho hamiltoniano, aplicando-a em um grafo predefinido, que é passado para a função por meio da sua matriz de adjacências.

Vale a pena compilar, executar e analisar esse programa.

Caminhos mínimo e máximo

Esses métodos aplicam-se a grafos valorados, nos quais temos valores associados aos vértices de cada nó do grafo.

Consideremos a [Figura 14.11](#), reproduzida a seguir:



Note que temos valores associados entre os nós representando distâncias entre cidades. Podemos, nesse caso, implementar algoritmos para determinar o menor e o pior caminho entre dois nós.

Para essas aplicações de grafos, temos numa série de algoritmos já desenvolvidos por cientistas e pesquisadores da ciência da computação e de outras

áreas das ciências exatas. Considerando que não é objetivo deste capítulo aprofundar, ou mesmo analisar e discutir modelos de algoritmos para determinação de caminhos em grafos, a seguir vamos apenas citar alguns deles.

- **Algoritmo de Dijkstra** - algoritmo para encontrar o caminho mínimo em grafo orientado ponderado, no qual o tamanho de cada aresta pode ser diferente. Um exemplo de aplicação desse é o Google Maps, onde se pode encontrar a melhor rota entre dois pontos;
- **Algoritmo de Prim** - algoritmo para encontrar a árvore de espalhamento mínimo ou árvore de espalhamento com custo mínimo de um grafo;



Papo técnico

Árvore de espalhamento é um subconjunto de arestas de um grafo, em que existe apenas uma aresta entre dois nós e em que todos os nós são conectados. Um exemplo de aplicação do algoritmo de Prim é a construção de uma rede de comunicação interligando cidades, a um custo mínimo.

- **Algoritmo de Floyd** - algoritmo para encontrar o caminho mínimo entre todos os pares de vértices de um grafo denso, usando o método da programação dinâmica.
- **Algoritmo de Kruskal** - algoritmo que constrói a árvore geradora de custo mínimo de um grafo adicionando arestas, uma de cada vez, a essa árvore. O conjunto de arestas forma uma floresta.
- **Algoritmo de Bellman-Ford** - esse algoritmo resolve o problema do caminho mais curto, de única origem, no qual os pesos das arestas podem ser negativos.

Enfim, depois desses conceitos e exemplos sobre grafos, você deve estar concordando com aquilo que escrevemos no início deste capítulo, ou seja, que o assunto *grafos* é interessante, abrangente... e complexo.

Pois, então, que tal aplicarmos aquilo que aprendemos até agora?
Vamos lá!



Vamos programar

Nesta seção, vamos mostrar a implementação dos Códigos 14.1, 14.2, 14.3 e 14.4 nas linguagens Java e Python, lembrando que os dois primeiros códigos implementam métodos de percurso ou busca, em largura e profundidade em grafos, e os dois últimos implementam o método e demonstram a determinação de caminho euleriano em um grafo.

Java

Código 14.1

```
// Procedimento de percurso ou busca em Largura, em grafos - (BFS - breadth first search)

public static void busca_largura(int inicio, int fim, int[] lista_nos, int[] vet_dista,
int[] cardP, int[][] lista_adj_P)
{
    int wk_no, ind_ln, n_ini, n_fim, ind_la, wk_la, indvet;
    n_ini = fim + 1;
    n_fim = fim;

    for (ind_ln = inicio; ind_ln <= fim; ind_ln++) {
        wk_no = lista_nos[ind_ln]; // escolhe primeiro nó do grafo para início do caminhamento

        for (ind_la = 1; ind_la <= cardP[wk_no]; ind_la++) // Percorre arestas do Nó escolhido
        {
            wk_la = lista_adj_P[wk_no][ind_la]; // Salva posição da aresta

            if (vet_dista[wk_la] == -1) // Verificada se nó selecionado possui aresta
            {
                vet_dista[wk_la] = vet_dista[wk_no] + 1; // Visita vértices adjacentes
                n_fim++;
                lista_nos[n_fim] = wk_la; // Marca nó visitado
            }
        }
    }

    if (n_ini <= n_fim) {
        busca_largura(n_ini, n_fim, lista_nos, vet_dista, cardP, lista_adj_P);
    }
}
```

Código 14.2

```
// Procedimento de percurso ou busca em Profundidade, em Grafos a partir da origem - (DFS - depth first search)

public static void busca_profundidade(int vertice, int[][] listAdjP, int[] mark, int[]
cardP)
{
    int w_vertice, ind;
    if (mark[vertice] == 0) // 0 - vértice não percorrido; 1 - vértice percorrido
    {
        System.out.println(" " + vertice); // Mostra Nó a ser percorrido
        mark[vertice] = 1; // Retira o vértice do grafo
        for (ind = 1; ind <= cardP[vertice]; ind++) {
            w_vertice = listAdjP[vertice][ind]; // Próximo Nó a ser visitado
            System.out.println(vertice + " -> " + w_vertice); // Mostra Nós percorridos
            busca_profundidade(w_vertice, listAdjP, mark, cardP); // Recursividade
        }
    }
}
```

Código 14.3

```
// Procedimento para determinar se um grafo possui um Caminho Euleriano

public static void caminho_euleriano(int[][] mat_adj, int nos)
{
    int qtd_nos_impar, tot_grau_nos, lin, col;
    qtd_nos_impar = 0;
    lin = 0;

    while ((qtd_nos_impar <= 2) && (lin < nos))
    {
        tot_grau_nos = 0;
        for (col=0; col < nos; col++)
        {
            tot_grau_nos = tot_grau_nos + mat_adj[lin][col];
        }

        if (tot_grau_nos%2 != 0)
            qtd_nos_impar++;
        lin++;
    }

    if (qtd_nos_impar > 2)
        System.out.println("Não existe um caminho Euleriano!");
    else
        System.out.println("Existe um caminho Euleriano!");
}
```

Código 14.4

```
//Determina se existe, ou não, caminho Euleriano no Grafo da Figura 14.11
public class JavaApplication2
{
    private static int MAXNOO = 6;
    private static int MAXNOD = 6;

    public static void main(String[] args) {

        int[][] mat_adjacencias = {{0, 1, 1, 1, 0, 0},
                                   {0, 0, 0, 1, 1, 0},
                                   {0, 0, 0, 1, 0, 0},
                                   {0, 0, 0, 0, 0, 1},
                                   {0, 0, 0, 0, 0, 1},
                                   {0, 0, 0, 0, 0, 1}};
        int fim, lin, col;

        System.out.println("Matriz de Adjacencias: ");
        fim = MAXNOD - 1;
        for (lin = 0; lin <= fim; lin++) {
            System.out.println();
            for (col = 0; col <= fim; col++) {
                System.out.println(mat_adjacencias[lin][col]);
            }
        }

        caminho_euleriano(mat_adjacencias, 6);    // Chamada do procedimento - Caminho Euleriano

        System.out.println();
        System.out.println(">>>> ");
    }

    public static void caminho_euleriano(int[][] mat_adj, int nos) {
        int qtd_nos_impar, tot_grau_nos, lin, col;
        qtd_nos_impar = 0;
        lin = 0;

        while ((qtd_nos_impar <= 2) && (lin < nos)) {
            tot_grau_nos = 0;
            for (col = 0; col < nos; col++) {
                tot_grau_nos = tot_grau_nos + mat_adj[lin][col];
            }

            if (tot_grau_nos % 2 != 0) {
                qtd_nos_impar++;
            }
            lin++;
        }

        if (qtd_nos_impar > 2) {
            System.out.println("Não existe um caminho Euleriano!");
        } else {
            System.out.println("Existe um caminho Euleriano!");
        }
    }
}
```

Python

Código 14.1

```
# Procedimento de percurso ou busca em Largura, em grafos - (BFS - breadth first search)

MAXNOD = 6          # Numero Maximo de Nos do grafo

def busca_largura(inicio, fim, lista_nos, vet_dista, cardP, lista_adj_P):
    # inicio e fim - no inicial e final do grafo
    # lista_nos - lista contendo os Nos de um Grafo
    # vet_dista - lista de arestas do grafo
    # cardP - vetor de cardinalidade do grafo
    # lista_adj_P - matriz de adjacências

    wk_no=0
    ind_ln=0
    n_ini=0
    n_fim=0
    ind_la=0
    wk_la=0
    indvet=0
    n_ini = fim + 1
    n_fim = fim
    ind_ln = inicio

    while ind_ln <= fim:
        wk_no = lista_nos[ind_ln]          # escolhe primeiro no do grafo para inicio do caminhamento
        ind_la = 1
        while ind_la <= cardP[wk_no]:
            wk_la = lista_adj_P[wk_no][ind_la];          # Percorre arestas do No escolhido
            # salva posição da aresta
            if vet_dista[wk_la] == -1:          # Verificada se no selecionado possui aresta
                vet_dista[wk_la] = vet_dista[wk_no] + 1          # visita vértices adjacentes
                n_fim+=1
                lista_nos[n_fim] = wk_la          # marca no visitado
                ind_la+=1
            ind_ln+=1
    # Chamada recursiva do procedimento
    if n_ini <= n_fim:
        busca_largura(n_ini, n_fim, lista_nos, vet_dista, cardP, lista_adj_P)
```

Código 14.2

```
# Procedimento de percurso ou busca em Profundidade, em Grafos a partir da origem - (DFS - depth first search)

MAXNOD = 6          # Numero maximo de Nos do grafo

def busca_profundidade(vertice, listAdjP, mark, cardP):

    # listAdjP - matriz de adjacências do grafo
    # mark - vetor com vértices percorridos
    # cardP - cardinalidade do grafo (Nos adjacentes)

    w_vertice=0
    ind=0
    if mark[vertice] == 0:                # 0 - vértice não percorrido; 1 - vértice percorrido
        print(" %d" %(vertice))          # Mostra No a ser percorrido
        mark[vertice] = 1;                # retira o vértice do grafo
        ind = 1
        while ind <= cardP[vertice]:
            w_vertice = listAdjP[vertice][ind]          # próximo No a ser visitado
            print(" %d -> %d " %(vertice,w_vertice))    # Mostra Nos percorridos
            busca_profundidade(w_vertice,listAdjP,mark,cardP) # recursividade
            ind+=1
```

Código 14.3

```
# Procedimento para determinar se um grafo possui um Caminho Euleriano

MAXNOO =3           # Numero máximo de Nós Origem no Grafo G
MAXNOD =3           # Numero máximo de Nós Destino no Grafo G

def caminho_euleriano(mat_adj,nos): # Recebe a Matriz de Adjacências do Grafo G e a quantidade de Nos do grafo

    qtd_nos_impar= 0
    tot_grau_nos= 0
    lin= 0
    col= 0
    fim= 0
    qtd_nos_impar = 0
    lin = 0

    while qtd_nos_impar <= 2 and lin < nos: # Quando quantidade de Nos Impares > 2 interrompe repetição
        tot_grau_nos = 0
        col=0
        while col < nos:
            tot_grau_nos = tot_grau_nos + mat_adj[lin][col] # Total de Graus dos Nós
            col+=1
```

```

        if tot_grau_nos%2 <> 0 :          # determina quantidade de Nos Impares
            qtd_nos_impar+=1
        lin+=1
    if qtd_nos_impar > 2:
        print("\n\n NAO existe um caminho Euleriano no Grafo G !")
    else:
        print("\n\n EXISTE um caminho Euleriano Grafo G !")

```

Código 14.4

```

#Determina se existe, ou não, caminho Euleriano no Grafo da Figura 14.11

MAXNOO =6          # Numero máximo de Nós Origem
MAXNOD =6          # Numero máximo de Nós Destino

def caminho_euleriano(mat_adj,nos):
    qtd_nos_impar= 0
    tot_grau_nos= 0
    lin= 0
    col= 0
    qtd_nos_impar = 0
    lin = 0

    while qtd_nos_impar <= 2 and lin < nos:
        tot_grau_nos = 0
        col=0
        while col < nos:
            tot_grau_nos = tot_grau_nos + mat_adj[lin][col]
            col+=1
        if tot_grau_nos%2 <> 0:
            qtd_nos_impar+=1
        lin+=1
    if qtd_nos_impar > 2:
        print("\n\n Nao existe um caminho Euleriano!");
    else:
        print("\n\n Existe um caminho Euleriano!");

#funcao principal

mat_adjacencias =
[[0,1,1,1,0,0],[0,0,0,1,1,0],[0,0,0,1,0,0],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1]]
fim=0
lin=0
col=0
print("Matriz de Adjacências: \n");
fim = MAXNOD - 1
lin = 0
while lin <= fim:
    col = 0
    while col <= fim:
        print(" %d " %(mat_adjacencias[lin][col]))
        col+=1
    lin+=1

caminho_euleriano(mat_adjacencias,6) # Chamada do procedimento - Caminho Euleriano

```



Para fixar

1. Determinado grupo de pessoas, em uma rede social, é constituído dos seguintes amigos: Ana, Mário, João, Mariana, Giuliana, Marcelo, Francisco. Pode-se utilizar um grafo para mostrar o relacionamento de amizade entre essas pessoas, de forma que os nós desse grafo representem as pessoas, e suas arestas ou arcos representem suas relações de amizade. Represente esse grafo e determine:
 - a. todos os amigos de Marcelo;
 - b. todos os amigos de Francisco;
 - c. todos os amigos de Mário.Você escolheu construir um grafo orientado ou não orientado? Justifique sua escolha.
2. Represente os grafos do exercício 1 usando:
 - a. uma matriz de adjacências;
 - b. uma lista de adjacências.
3. Determine o grau de cada nó do grafo (b) do exercício 1 e, em seguida, determine o grau desse grafo.



Para saber mais

Você está no penúltimo capítulo deste livro e, com certeza, entendeu a importância do estudo das estruturas de dados dentro da área de algoritmos e programação de computadores. Esse é o propósito geral deste trabalho, ou seja, inculcar, de forma clara e exemplificada, os conceitos básicos sobre estruturas de dados, bem como algumas formas de manipulá-las. Se você deseja aprofundar os estudos sobre grafos, sugerimos a pesquisa em livros sobre estrutura de dados, algoritmos, e algoritmos e estruturas de dados. Para isso, recomendamos as seguintes leituras:

- Capítulo 6 do livro *Algoritmos: teoria e prática* (Cormen et al., 2012);
- Capítulo 16 do livro *Estruturas de dados e algoritmos: padrões de projetos orientados a objeto com Java* (Preiss, 2000) que apresenta métodos e algoritmos de busca em grafos, desenvolvidos por alguns autores, usando o conceito do caminho mais curto.



Navegar é preciso

Na internet existem vários sites que apresentam importantes materiais sobre grafos, os quais permitem um aprofundamento nos estudos a respeito desse tema.

A organização Boost disponibiliza uma vasta biblioteca de grafos para desenvolvedores de programas em C. Para saber mais sobre essa biblioteca, você pode acessar o link: www.boost.org ou consultar o livro *The Boost Graph Library* (Siek *et al.*, 2002).

No site da Wikipedia (<http://pt.wikipedia.org/wiki/Grafos>) você encontrará mais informações interessantes relacionadas a grafos. O site aborda o tema em outras áreas, não apenas em computação.

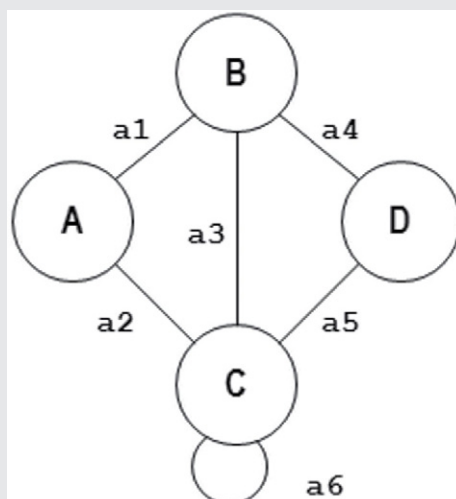
O site Learn you some Erlang (<http://learnyousomeerlang.com/a-short-visit-to-common-data-structures#directed-graphs>), conforme já mencionado em capítulos anteriores, também aborda o tema *grafos orientados*.

No link <http://www.din.uem.br/sbpo/sbpo2009/artigos/55491.pdf> você encontrará um artigo intitulado *Teoria de grafos: uma proposta de objeto de aprendizagem para o modelo EAD*. Nesse artigo, o autor pesquisou três objetos de aprendizagem relacionados com a teoria dos grafos para embasar seus estudos e propor de um objeto denominado AlgoDeGrafos. Sem dúvida, vale a pena ler e efetuar o *download* de algumas dessas ferramentas, para usar e analisar sua aplicabilidade.

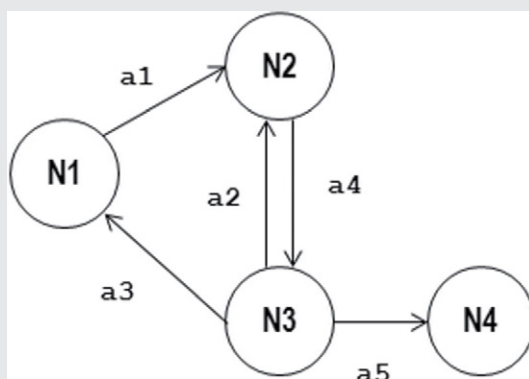
Exercícios

1. Em uma prova bimestral, na disciplina Algoritmos, um aluno afirmou que os grafos (a) e (b), abaixo, são, respectivamente, orientado e não orientado. Você concorda com esse aluno? Justifique sua resposta.

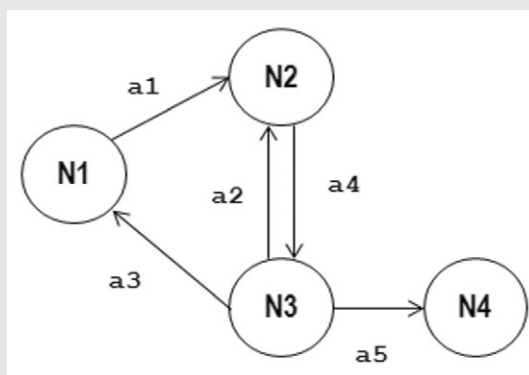
a)



b)



2. Considerando a [Figura 14.3](#), determine quantos vértices são adjacentes ao nó 4, e, considerando a [Figura 14.5](#), determine quantos vértices são adjacentes ao nó E.
3. Considerando o grafo a seguir, construa uma matriz de adjacências e uma de incidência para representar os relacionamentos entre seus nós.



Glossário

OSI/RM: modelo de referência (**R**eference **M**odel) para interconexão de sistemas abertos ou não proprietários (**O**pen **S**ystems **I**nterconnection), disponibilizado pela ISO (**I**nternational **O**rganization for **S**tandardization, em português: **O**rganização **I**nternacional para **P**adronização). O modelo contém uma descrição, e não um novo padrão, que sugere o modo como dados e informação devem ser transmitidos entre os nós (ou pontos) de uma rede de computadores. Ele é estruturado em sete camadas (Aplicação, Apresentação, Sessão, Transporte, Rede, Enlace de dados e Física), e em cada uma delas estão contidos normas e procedimentos para o desenvolvimento

de hardware e software (protocolos) que funcionam sob qualquer plataforma computacional, independentemente do hardware ou software utilizado.

Protocolo de rede: conjunto de regras, expressas em algoritmos, que regem o funcionamento de unidades funcionais de um sistema de comunicação e que devem ser seguidas para que a comunicação seja alcançada.

Roteadores ou routers: equipamento utilizado em redes de computadores, cuja função principal é interconectar redes distintas e efetuar o encaminhamento de pacotes de dados que trafegam entre elas.

Referências bibliográficas

- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- KOFFMAN, E. B.; WOLFGANG, P. A. T. *Objetos, abstração, estruturas de dados e projeto usando C++*. Trad. Sueli Cunha. Rev. Tec. Orlando Bernardo Filho, João Araújo Ribeiro. Rio de Janeiro: LTC, 2008.
- PREISS, B. R. *Estruturas de dados e algoritmos: padrões de projetos orientados a objeto com Java*. Trad. Elizabeth Ferreira. Rio de Janeiro: Campus, 2000.
- SIEK, J.; LEE, L.; LUMSDAINE, A. *The Boost Graph Library*. Boston: Addison-Wesley, 2002.
- SILVA, O. Q. *Estrutura de dados e algoritmos usando C - fundamentos e aplicações*. Rio de Janeiro: Ciência Moderna, 2007.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.
- VELOSO, P. et al. *Estruturas de dados*. Rio de Janeiro: Campus, 1983.



CO QUE VEM DEPOIS

Ufa! Estamos quase chegando ao último capítulo deste livro. Certamente, você já percebeu a quantidade de informações sobre estruturas de dados e suas aplicações que recebeu até este momento. Considerando que a informação é a matéria-prima para a produção do conhecimento, acreditamos que, com base nas informações que transmitimos até o momento e em seus conhecimentos anteriores, você gerou novo conhecimento sobre algoritmos e, principalmente, estruturas de dados.

Podemos pensar dessa forma?

Se você não tiver certeza disso, talvez deva revisar alguns conceitos, ou mesmo exemplos e exercícios contidos nos capítulos anteriores, ou até mesmo nas referências bibliográficas e *links* sugeridos em cada um desses capítulos.

Agora, se você percebe que agregou conhecimentos suficientes para resolver problemas usando estruturas de dados, chegou o momento da sua autoavaliação, pois, no Capítulo 15, vamos dar alguns exemplos de aplicações sobre estruturas de dados, sugerindo que você desenvolva soluções para eles.

Preparado? Então, bons estudos e sucesso nos projetos que serão sugeridos!

Aplicações

“As ideias são como as nozes, e, até hoje, não descobri melhor processo para saber o que há dentro de umas e outras, senão quebrá-las.”

MACHADO DE ASSIS

A informação é a matéria-prima da qual se extrai o conhecimento, por meio do qual adquirimos sabedoria. Por isso, podemos afirmar que, enquanto o conhecimento e a sabedoria nos permitem ter novas ideias, o experimento é a forma mais indicada de testá-las. Experimentar é “quebrar as nozes”.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- entender a importância do conhecimento obtido por meio dos conceitos explicitados nos capítulos anteriores;
- projetar e implementar algoritmos e programas, tendo esses conceitos como base, para solucionar problemas computacionais;
- pensar e criar aplicações com base nos exemplos e exercícios sugeridos em cada capítulo.

Como você sabe, muitos dos problemas computacionais resolvidos por meio da construção de algoritmos/programas resultam de ideias ou mesmo de abstrações de situações do mundo real. Com base em ideias, nós construímos modelos mentais lógicos para, em seguida, aplicando os conceitos e teorias sobre construções lógicas de algoritmos, implementar modelos de algoritmos que serão executados pelo computador para solucionar problemas.

Até este momento, diversos assuntos novos – ao menos para alguns leitores – foram apresentados e estudados, e isso nos possibilitou conhecer, ou até mesmo reforçar, um número razoável de conceitos e teorias sobre a construção de algoritmos/programas. Agora só nos resta experimentar isso tudo, abstraindo, projetando, desenvolvendo e implementando sistemas, ou *aplicações*, como muitos profissionais denominam os sistemas.

Como afirma o professor José Augusto Fabri, da Universidade Federal Tecnológica do Paraná, “O sistema está ligado a alguma coisa nova, a uma nova teoria, a uma invenção, a um novo paradigma”.



Para começar

Ao longo de seus estudos, principalmente nas disciplinas Algoritmos e Programação, quantas vezes você se debruçou sobre os problemas sugeridos e tentou encontrar soluções para eles?



Certamente, inúmeras vezes e sempre recorrendo a teorias, conceitos e exemplos que adquiriu – e vem adquirindo – nesses anos de estudos.

Na construção de algoritmos e programas de computador, quanto mais aplicamos aquilo que aprendemos, mais sedimentados ficam os conceitos, as sintaxes, as estruturas lógicas, os modelos etc. Portanto, o desenvolvimento de novos projetos, sistemas ou aplicações nos ajuda a entender e a solucionar problemas, possibilitando a integração da teoria com a prática.

Vamos experimentar?



Conhecendo a teoria para programar

Até o momento, você estudou vários assuntos que foram distribuídos entre os seguintes capítulos:

1. Tipos abstratos de dados (TAD).
2. Mapa de memória de um processo.
3. Recursão.
4. Métodos de busca.
5. Métodos de ordenação.
6. Alocação dinâmica.
7. Listas ligadas lineares.
8. Outros tipos de lista.
9. Filas.
10. Pilhas
11. Árvores
12. Árvores N -árias.
13. Árvores balanceadas.
14. Grafos.

Os conteúdos desses capítulos serão fundamentais durante a construção de algoritmos e programas para solucionar problemas computacionais com maior nível de complexidade lógica em relação àqueles que você estava acostumado a desenvolver.

Na construção de programas de computadores, recomendamos recapitular o conteúdo do “Para começar” do último capítulo do livro *Algoritmos e programação de computadores* (PIVA, D. *et al.*, 2012). Nele estão pontuadas algumas propriedades de um bom sistema, também válidas para um programa de computador, que reproduzimos e enumeramos a seguir:

- **ser eficiente e eficaz** - dar respostas ao usuário em tempo adequado, executando de forma correta e lógica suas instruções;

- **ser portátil** - poder ser executado em diversas plataformas computacionais, por computadores que utilizam sistemas operacionais diferentes;
- **ser seguro e tolerante a falhas** - não permitir sua utilização por qualquer usuário, nem deixar as conhecidas *backdoor*, que permitem a entrada e a instalação de vírus;
- **confiável e disponível** - processar dados e informações de forma consistente; quando em execução, estar disponível o tempo todo que o computador permanecer ligado, seja onde estiver sendo executado;
- **modular e escalável** - permitir modificações sempre que ocorrerem mudanças ambientais, por meio da incorporação de novos processos ou funções, sem afetar seu desempenho.

Em “Vamos programar”, a seguir, sugerimos novos exemplos de aplicações que abrangem os conceitos apresentados em cada capítulo. Você, certamente, poderá desenvolver alguns deles ou, se preferir, todos. De qualquer forma, esperamos que você os desenvolva considerando – e aplicando – algumas dessas características – ou todas elas. Para tanto, é recomendável a escolha de uma boa linguagem de programação.

Opá! E quais são as características de uma boa linguagem de programação?

Existem inúmeras, porém, para propósitos gerais, podemos considerar que uma boa linguagem de programação é aquela que:

- possui extensiva capacidade de gerenciamento de arquivos e bancos de dados com rapidez, integridade e segurança;
- trata arquivos independentemente da quantidade de registros, sem perder *performance*;
- tem compatibilidade com outras bases de dados;
- tem portabilidade, ou seja, pode ser executada em qualquer plataforma de máquina, sem necessidade de modificações ou recompilações;
- pode ser utilizada em vários ambientes computacionais e por diferentes IDE (*Integrated Development Environment*);
- transmite confiabilidade e mantém sua continuidade (disponibilidade de novas versões);
- proporciona, por intermédio do fabricante, bom suporte técnico, e dispõe de vasta e boa documentação técnica e literária nos mais diversos suportes midiáticos;
- tem uma interface gráfica amigável (GUI).

Enfim, uma boa linguagem de programação é aquela que satisfaz melhor as necessidades do programador de computador e está adequada para a solução dos problemas que se apresentam.

Por fim, o mais importante é pensar em todos os problemas sugeridos como um novo desafio para sua criatividade.

Considerando que são os desafios que nos movem, vamos em frente!



Vamos programar

Um dos principais objetivos da construção de algoritmos e da programação de computadores é desenvolver a capacidade de raciocínio lógico, abstração, interpretação e análise crítica de dados e informações, criando e implementando soluções computacionais para problemas previamente identificados.

Para a implementação dos algoritmos exemplificados em cada capítulo, utilizamos três diferentes linguagens de programação. Todavia, a construção de modelos lógicos para a solução de problemas computacionais independe da linguagem de programação utilizada, pois, para escrever algoritmos, basta dispor de lápis e papel.

Sugerimos, a seguir, alguns problemas do cotidiano, propondo algumas estratégias para seu desenvolvimento. Fica a seu critério a escolha da linguagem de programação para sua implementação. Todavia, a escolha do TAD ou estrutura de dados, bem como da estrutura lógica mais adequada para a solução do problema, dependerá dos conhecimentos que você obteve nos capítulos anteriores. Vamos lá!

1. Uma estrutura de dados muito utilizada na área da ciência da computação é a fila. Dentre as muitas aplicações dos conceitos e teorias sobre filas, destacam-se aqueles utilizados na construção de sistemas operacionais. Como sabemos, o sistema operacional tem várias funções, que, *grosso modo*, podem ser divididas em duas categorias: máquina virtual e gerenciador de recursos. Enquanto máquina virtual, ele fornece a base sobre a qual os programas de aplicação são escritos e executados, e, como gerenciador de recursos, sua função primordial é controlar ou gerenciar a utilização de todos os recursos fornecidos pelo *hardware* do computador e, principalmente, sua distribuição entre os diversos programas (processos) que competem (disputam) por ele. Como você pode perceber, estamos falando de controlar prioridades.

Então, sugerimos que você elabore uma aplicação que controle as tarefas (processos) que esperam por um recurso de *hardware* escasso, que são as impressoras. Isso mesmo: esse aplicativo deve controlar os pedidos de uso de impressora por processos, na ordem em que forem colocados em execução, ou seja, obedecendo a uma ordem prioritária.

Você pode pensar em implementar o seguinte:

- a. listar todos os processos que estão aguardando liberação da impressora;
- b. informar o tempo de espera de cada processo na fila de impressão;
- c. permitir mudar a prioridade de execução de um processo;
- d. permitir retirar (cancelar) um processo da fila.

Se abstrair um pouco mais sobre esse exercício, talvez você descubra mais alguma funcionalidade para esse aplicativo.

2. Uma empresa agropecuária dispõe, entre outros produtos que vende, daqueles que têm prazo de validade. Se esses prazos não forem controlados, a empresa poderá perder muitos produtos com prazo de validade vencido ou até mesmo sofrer processos judiciais, caso venda um produto nessa condição. Portanto, um aplicativo que controla prazos de validade é estratégico nesse tipo de negócio, assim como em outros similares.

Gostaria de ajudar essa empresa a resolver tal problema?

Se a resposta for SIM, desenvolva um aplicativo para controlar o prazo de validade de produtos que entram no estoque de produtos acabados de uma empresa.

Sugestão:

Imagine que a empresa precisa que, além de controlar o prazo de validade, esse aplicativo forneça informação importante, como: quantidade de produtos em estoque; valor total do estoque de produtos com prazo de validade a expirar em determinado tempo (daqui a 7 dias, um mês ou outro qualquer, informado previamente). E então? Você imaginou que terá de trabalhar com uma estrutura de dados heterogênea (registro) ligada a outras estruturas homogêneas?

Tem mais: a empresa também poderia pedir que essa aplicação permitisse visualizar os produtos em estoque por descrição, por valor e, obviamente, por prazo de validade. Nesses casos, você estaria aplicando os métodos de ordenação, aquilo que aprendeu no Capítulo 5 deste livro.

3. Uma cidade costuma ser dividida em regiões considerando-se os pontos cardeais, ou seja, norte, sul, leste e oeste. Olhe o mapa da cidade onde você mora e procure identificar essa divisão. Em seguida, localize as instituições públicas, ou mesmo privadas (entidades), que são de interesse da população que fazem parte de cada região: hospitais, delegacias, corpo de bombeiros, escolas, creches, etc.

Agora, que tal construir uma aplicação que contemple, por região, uma relação de entidades, com seus endereços e telefones de emergência?

Vamos lá!

Sugestões:

- a. possibilitar inserção e exclusão de entidades;
 - b. possibilitar a alteração de dados sobre cada entidade;
 - c. possibilitar a busca de informações sobre essas entidades;
 - d. possibilitar a ordenação dos dados sobre as entidades, por exemplo, por bairro, logradouro, nome, telefone, etc.
4. A principal atividade de um carteiro é entregar correspondências postais. Você pode notar que todos eles carregam uma sacola com pilhas de correspondências. Quando um carteiro chega a determinado endereço, ele retira e entrega a correspondência que está no topo da pilha. Considerando isso, elabore um programa para simular os processos executados por um carteiro. Ao pensar esse programa, você certamente estará aplicando os conceitos sobre alocação dinâmica de memória (Capítulo 6) e pilhas (Capítulo 10). Lembre-se de que a pilha é uma estrutura de dados com a característica de acessibilidade somente ao elemento que está no topo.

Vamos lá!

Você pode pensar nos seguintes processos e torná-los computacionais:

- a. inserir correspondência na sacola (empilhar);
- b. remover correspondência da sacola (desempilhar);
- c. verificar se a sacola está cheia ou vazia;
- d. verificar a próxima correspondência a ser entregue;
- e. informar quantas correspondências existem na sacola;

Feito isso, que tal ajudar o carteiro a traçar o caminho mínimo para a entrega das correspondências – afinal, todas as correspondências têm um endereço/destino. Lembre-se dos conceitos relativos a caminhoamento mínimo em um grafo? Este é o momento de aplicá-lo.

5. Uma aplicação interessante para o exercício dos conceitos relacionados a pilhas é a compilação de uma expressão aritmética. Quando você estudou operadores aritméticos, aprendeu sobre a prioridade desses operadores dentro de uma expressão, não é mesmo? Então, que tal escrever um programa que compile uma expressão aritmética, como:

$a - b / e + c * d$

Critérios:

- a) os operadores da expressão aritmética são representados pelos caracteres: *, /, + e -;
- b) os operandos são representados por letras de *a* até *j*;
- c) os resultados das operações são representados (armazenados) por letras, de trás para a frente, iniciando-se em *z*;

A tabela a seguir representa o resultado da compilação da expressão modelo.

Operador	Operando 1	Operando 2	Resultado
/	<i>b</i>	<i>e</i>	<i>Z</i>
*	<i>c</i>	<i>d</i>	<i>Y</i>
-	<i>a</i>	<i>z</i>	<i>x</i>
+	<i>x</i>	<i>y</i>	<i>v</i>

Procedimentos:

- a) o programa deve receber os caracteres e as letras especificados na tabela;
- b) o programa deve validar esses caracteres, ou seja, se o caractere for um espaço em branco, você deverá ignorá-los; todavia, se não for nenhum dos caracteres da tabela, envie mensagem de erro e encerre o programa;
- c) se o caractere for um operando, coloque-o na pilha de operandos;
- d) se o caractere for um operador, então compare sua precedência (prioridade) com o operador que está no topo da pilha de operadores;
- e) se o operador em questão (atual) tem prioridade maior do que aquele que está no topo da pilha, ou a pilha for NULL, então o operador atual deve ir para o topo da pilha;
- f) se o operador atual tem a mesma prioridade ou prioridade menor, então o operador do topo da pilha é o próximo a ser avaliado, da seguinte forma: retira-se o operador da pilha de operadores, com um par de operandos, que está na pilha de operandos, e monta-se uma nova linha na tabela de saída. O caractere escolhido para armazenar o resultado da operação dessa linha deve ser colocado na pilha de operandos;
- g) Após isso, o operador atual deve ser novamente comparado com aquele que ficou no topo da pilha de operadores. Repita esse processo até que a pilha de operadores seja NULL.

Após desenvolver e testar essa aplicação, que tal você pensar em modificá-la para implementar parênteses nas expressões aritméticas?

Lembre-se de que, em expressões, os parênteses são utilizados para modificar a precedência (ou prioridade) dos operadores. Temos aí um novo desafio para você. Aceita?

Bem, até aqui você avançou muito nas aplicações sobre estrutura de dados ou TAD, listas, filas, pilhas. Portanto, as próximas sugestões de aplicações relacionam-se com árvores, que, conforme estudamos nos Capítulos 11, 12 e 13, são estruturas de dados baseadas em listas encadeadas que têm um nó superior (também chamado de raiz) que aponta para outros nós (chamados de nós filhos), que também podem ser pais de outros nós.

Então, vamos sugerir uma aplicação para esses conceitos.

6. Você já ouviu falar em árvore genealógica de uma família?

Conceitualmente: “Uma árvore genealógica é um histórico de certa parte dos antepassados de um indivíduo ou família. Mais especificamente, trata-se de uma representação gráfica genealógica para mostrar as conexões familiares entre indivíduos, trazendo seus nomes e, algumas vezes, datas e lugares de nascimento, casamento, fotos e falecimento.” (Wikipedia)

Que tal construir a árvore genealógica da sua família iniciando-a pelo seu bisavô. Essa é uma boa aplicação dos conceitos relativos a árvores e a listas, pois você poderá armazenar dados sobre as pessoas da família e, portanto, aplicar os conceitos estudados nos Capítulos 7 e 8.

Após montar a árvore genealógica de sua família, desenvolva uma aplicação para implementá-la computacionalmente.

Você pode pensar nos seguintes processos e torná-los computacionais:

- a. inserir um novo nó filho nessa árvore quando alguém nascer;
- b. excluir um nó dessa árvore quando alguém falecer;
- c. inserir nós quando ocorrer um casamento;
- d. efetuar buscas na árvore a partir de determinado nó (família ou pessoas).

Em todos os casos, você vai aplicar os conceitos referentes à inserção de nós à esquerda e à direita, e também ao percurso em uma árvore.

Você terá que rotular os nós e criar listas ligadas com mais informações sobre cada família ou mesmo integrantes da família.

E então, está pronto para pensar esse problema?

Nossas próximas sugestões de aplicações referem-se ao assunto *grafos*.

Já vimos que grafos armazenam e facilitam a manipulação de estruturas de dados com características de ligações entre pontos (nós ou vértices).

Quando elaboramos algoritmos aplicando conceitos sobre grafos, certamente utilizamos estruturas de dados, como vetores, listas encadeadas ou matrizes. O uso de listas geralmente proporciona acesso mais rápido aos dados da estrutura; todavia, na maioria dos casos, é mais fácil utilizar vetores. Podemos verificar isso na literatura, cuja maioria dos exemplos utiliza-se de vetores.

Vamos para mais uma aplicação?

7. Imagine que a prefeitura de sua cidade está projetando um novo bairro, em um local de relevo bem plano e que permite ótimo planejamento urbano, principalmente das quadras e ruas. Entretanto, um dos desafios da prefeitura é planejar esse bairro de forma que ele tenha o maior número possível de ruas com sentido único.

Sugestões:

- a. procure saber quais ruas podem ter mão-única e para que sentido ela apontará, mantendo todas as ruas alcançáveis de qualquer outro ponto na cidade;
 - b. elabore e implemente um grafo orientado e rotulado, de forma que você possa, partindo de determinado ponto, chegar a um destino através de um caminho mínimo e um caminho máximo.
8. Um dos problemas enfrentados por uma pessoa numa grande cidade é localizar um restaurante próximo do ponto onde ela se encontra, para almoçar ou jantar. Atualmente, esse tipo de aplicação está disponível em dispositivos móveis (plataforma *Mobile*), que combinam informações de GPS com dados e informações sobre esses estabelecimentos, contidos em guias turísticos ou mesmo na lista telefônica.

Sugerimos que você construa uma aplicação que auxilie uma pessoa a encontrar um restaurante próximo do local onde ela está.

Sugestão:

Delimite uma área de cobertura (abrangência) para seu aplicativo. Considerando essa área, mapeie nela os principais restaurantes em funcionamento e estabeleça a distância entre eles. Com essas informações, você já pode pensar em construir um grafo e sua matriz de adjacências ou lista de incidências. Agora deve preocupar-se em escolher o método de caminhamento nesse grafo, a partir de um ponto (nó) inicial.

Ah! lembre-se de coletar e registrar outras informações sobre cada estabelecimento, tais como: telefone, e-mail, tipos de cartões aceitos, horários de funcionamento, etc. Armazene tudo isso em estruturas de dados, para que possam ser acessadas pelos usuários do aplicativo.

Outra sugestão de importante aplicação da teoria de grafos é encontrada na biologia e está relacionada com o sequenciamento da cadeia de

DNA. A cadeia de DNA é composta pelas bases A (adenina), C (citossina), G (guanina) e T (timina). Uma das aplicações de computadores nessa área (bioinformática) é o estabelecimento de medidas para similaridade entre sequências biológicas.

9. Considere como exemplos as sequências CATT e GAT. O possível alinhamento dessas sequências é dado a seguir.

G A - T T - C A - T

(-) representa uma falha.

Cada membro de uma sequência é emparelhado com um membro de outra sequência ou com uma falha, sendo atribuído um peso (valor) a cada emparelhamento.

(1) representa o emparelhamento de dois membros iguais.

(-1) representa o emparelhamento de dois membros diferentes.

(-2) representa o emparelhamento de um membro e uma falha.

A “qualidade” do alinhamento é dada pela soma dos valores (pesos) de todos os seus emparelhamentos.

No exemplo acima, o valor será: $-2 - 1 - 2 - 2 + 1 = -6$.

Agora, no alinhamento a seguir:

G A T T C A T -

temos os valores: $-1 + 1 + 1 - 2 = -1$.

Portanto, conforme esse critério, o segundo sequenciamento é de melhor qualidade. A evolução das técnicas de clonagem de moléculas, aliada aos avanços das tecnologias da informação e à evolução das técnicas de armazenamento e recuperação de dados e informações (bancos de dados), a partir de grandes bases de dados, vem exigindo cada vez mais aplicações (programas) de métodos de comparação de sequências biológicas.

Esse problema, e outros, pode ser resolvido aplicando-se os conceitos sobre grafos. Lembra-se da matriz de adjacências? Então, você pode construir um dígrafo correspondente ao problema de alinhamento das sequências acima, montar uma matriz de adjacências e, a partir daí, calcular o melhor alinhamento (o de valor mais alto) das sequências. Para isso, você deve achar o caminho mais longo (máximo), em que seu comprimento seja a soma dos comprimentos dos arcos no caminho.

Tente fazer isso!



Para saber mais...

Dos salões paroquiais às salas de aulas, destas para as salas de reuniões nas empresas, das salas de reuniões aos encontros sociais nos fins de semana, das redes privadas às redes sociais, sem dúvida nenhuma uma revolução está acontecendo, apoiada por tecnologias da informação. Temos hoje o conceito de “um para um” indo em direção ao de “um para muitos”, em um ambiente social, público e compartilhado. Isso é, bem superficialmente, o conceito das redes sociais, de que você certamente faz parte.

Atualmente, não somente as pessoas, mas também as empresas estão investindo em modelos de sistemas que permitam entender melhor os relacionamentos que ocorrem nessas redes e, principalmente, as informações que circulam nelas. Investir em aplicativos que auxiliem na análise de redes sociais deixou de ser uma necessidade dos departamentos de Marketing e Vendas para se tornar um assunto relacionado com estratégias empresariais. Isso ficou nítido com a criação do cargo de Analista de Redes Sociais.

Nesse contexto, podemos inferir sobre a importância dos assuntos abordados neste livro, notadamente aqueles relativos a árvores e grafos, e como estes podem ser utilizados para fazer a análise e o estudo das relações em redes, auxiliando pessoas e empresas a tirar proveito delas.

Para saber mais sobre isso, sugerimos que você acesse o link <http://labspace.open.ac.uk/course/view.php?id=4951&topic=all>, onde você vai encontrar textos sobre redes sociais e sobre a teoria de grafos aplicada às redes sociais. Obviamente, centenas de outros materiais sobre isso estão disponíveis em diversos suportes midiáticos, entretanto, este é de fácil leitura e entendimento.

Vamos ler?



Navegar é preciso

Percebeu quantas aplicações podemos pensar e solucionar usando algoritmos e, conseqüentemente, a programação de computadores?

Isso mesmo! Quanto mais abstraímos o mundo em que vivemos, seja muito próximo ou mesmo distante de nós, mais descobrimos problemas que podem ser solucionados usando o computador.

Você já deve ter ouvido falar, lido sobre o assunto ou até mesmo participado das Maratonas de Programação. A cada edição dessa competição,

seja em nível regional, nacional, seja em nível internacional, novos desafios (problemas computacionais) são lançados e as equipes participantes devem construir programas para vencê-los/solucioná-los. Portanto, sugerimos que você acesse o link <http://uva.onlinejudge.org/>, em que poderá ter acesso a vários problemas dados nessas maratonas.

No link <http://maratona.ime.usp.br/preparacao13.html> você encontrará outros sublinks importantes com problemas sugeridos nas maratonas da Sociedade Brasileira de Computação e também na Olimpíada Brasileira de Informática.

Vale a pena navegar nessa página!

A área de computação vem, ao longo da sua evolução, criando termos e terminologias que, muitas vezes, guardam em si ou provocam o desenvolvimento de aplicações complexas. Atualmente, ouve-se falar muito em *big data*. O que seria isso?

Em tecnologia da informação, *big data* é um conceito no qual os focos são o grande armazenamento de dados e a maior velocidade na recuperação desses dados. Podemos dizer que o *big data* baseia-se em 5V: *velocidade*, *volume*, *variedade*, *veracidade* e *valor*.

Se antes as empresas pecavam pela falta de dados e informações, hoje pecam pelo excesso, ou seja, potentes *data warehouse* armazenam *exabytes* de dados e informações, e sabe-se que o conhecimento tão necessário para a condução dos negócios das empresas está contido nas relações entre dados e informações. Entretanto, o maior problema enfrentado pelas empresas atualmente é como extrair o conhecimento de um *big data*.

Se você quer saber um pouco mais sobre como os conceitos de estruturas de dados vêm sendo aplicados nesse assunto ou mesmo em bioinformática, acesse os links a seguir:

<http://www.inf.ufpr.br/jbdorr/texto-quali-jbdorr.pdf>

<http://www.bibliotecadigital.unicamp.br/document/?code=000201497>

Para saber mais sobre *big data*, acesse:

https://www.ibm.com/developerworks/community/blogs/ctaurion/entry/voce_realmente_sabe_o_que_e_big_data?lang=en

No link http://mitpress.mit.edu/sites/default/files/titles/content/9780262033848_Solutions_to_Exercises_and_Problems.pdf você encontrará solução para exercícios contidos no livro *Algoritmos: teoria e prática* (Cormen, 2012).

Glossário

Bioinformática: área da ciência da computação que visa analisar dados e resolver problemas de aplicações biológicas por meio do desenvolvimento de ferramentas. O avanço tecnológico dos computadores possibilitou à bioinformática desenvolver softwares para processar grande volume de dados gerados pelos projetos da biologia.

(<http://www.ic.unicamp.br/~zanoni/orientacoes/mestrado/mangelica/PropostaMariaAngelica.pdf>)

GPS (*Global Positioning System*): sistema de navegação por satélite que fornece a um aparelho receptor móvel sua posição, assim como informação horária, sob todas as condições atmosféricas, a qualquer momento e em qualquer lugar na Terra, desde que o receptor se encontre no campo de visão de quatro satélites GPS. (Wikipedia)

IDE (*Integrated Development Environment*): programa de computador que reúne características e ferramentas de apoio ao desenvolvimento de software com o objetivo de agilizar esse processo. Geralmente, os IDE facilitam a técnica de RAD (*Rapid Application Development*, ou Desenvolvimento Rápido de Aplicativos), que visa a maior produtividade dos desenvolvedores. Um IDE pode incluir vários utilitários, todavia, todos implementam pelo menos editor de texto, compilador, *link-editor* ou montador, depurador e *run-time* (executor). (Wikipedia)

Mídias sociais: sistemas online que permitem a interação a partir da criação e do compartilhamento de conteúdo nos mais diversos formatos. Esses sistemas possibilitam a publicação de conteúdo por qualquer pessoa/empresa, baixando para quase zero os custos com produção e distribuição de algum material de divulgação. (Wikipedia)

Processo: sequência sistemática de operações que visam a produzir um resultado específico. Em computação, é um conjunto de instruções sequenciado que pode ser utilizado em um ou mais pontos, em um ou mais programas de computador, e que geralmente tem um ou mais parâmetros de entrada, produzindo um ou mais parâmetros de saída.

Sistema: conjunto de elementos que se conectam, harmonicamente, com o objetivo de formar um todo unificado. (Ludwig von Bertalanffy)

Software aplicativo (aplicativo ou aplicação): programa de computador que tem por objetivo ajudar seu usuário a desempenhar uma tarefa específica, em geral ligada ao processamento de dados. (Wikipedia)

Referências bibliográficas

- BATES, B.; SIERRA, K. *Use a cabeça: Java*. Rio de Janeiro: Alta Books, 2007.
- CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.
- LAFORE, R. *Estrutura de dados & algoritmos em Java*. Trad. Eveline Vieira Machado. Rio de Janeiro: Ciência Moderna, 2004.
- LISKOV, B.; ZILLES, S. Programming with Abstract Data Types. *ACM SIGPLAN Notices*, ano 9 (4) 50-59, 1974.
- MOKARZEL, F. C.; SOMA., N. Y. *Introdução à ciência da computação*. Rio de Janeiro: Elsevier, 2008.
- PIVA, D. et al. *Algoritmos e programação de computadores*. Rio de Janeiro: Campus, 2012.
- SCHILDT, H. C. *completo e total*. 3. ed. São Paulo: Makron Books, 1996.
- TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.
- VELOSO, P. et al. *Estruturas de dados*. Rio de Janeiro: Campus, 1983.
- WIRTH, N. *Algoritmos e estruturas de dados*. Rio de Janeiro: Prentice-Hall, 1989.



QUE VEM DEPOIS

Ufa! Chegamos ao final do nosso passeio pelas estruturas de dados.

Se você teve a oportunidade de ler nosso livro *Algoritmos e programação de computadores*, lançado em 2012 pela Editora Elsevier, então, com mais este livro, você certamente avançou muito e agregou conhecimentos sobre o assunto *algoritmos*. Diante disso, acreditamos que você está mais seguro e tem mais habilidades para construir aplicações com maior nível de complexidade de algoritmos. Aliás, a complexidade de algoritmos é o próximo assunto que sugerimos que você pesquise e estude, pois, dessa forma, avançará ainda mais na programação de computadores, encontrando soluções para problemas cada vez mais intrincados.

O poeta francês Jean Cocteau escreveu a seguinte frase: “*Não sabendo que era impossível, foi lá e fez*”. A um hábil construtor de algoritmos e programas, caberia essa frase com uma pequena modificação: “*Sabendo que era impossível, foi lá e fez*”. Assim deve agir um bom programador de computadores.

Sucesso!